

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Conventional software management practices are sound in theory, but practice is still tied to archaic (outdated) technology and techniques.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

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

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

All three analyses reached the same general conclusion: The success rate for software projects is very low.

The three analyses provide a good introduction to the magnitude of the software problem and the current norms for conventional software management performance.

1.1 THE WATERFALL MODEL

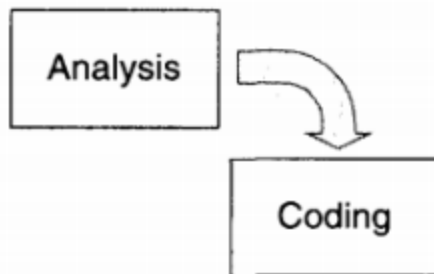
Most software engineering texts present the waterfall model as the source of the "conventional" software process.

1.1.1 IN THEORY

It provides an insightful and concise summary of conventional software management

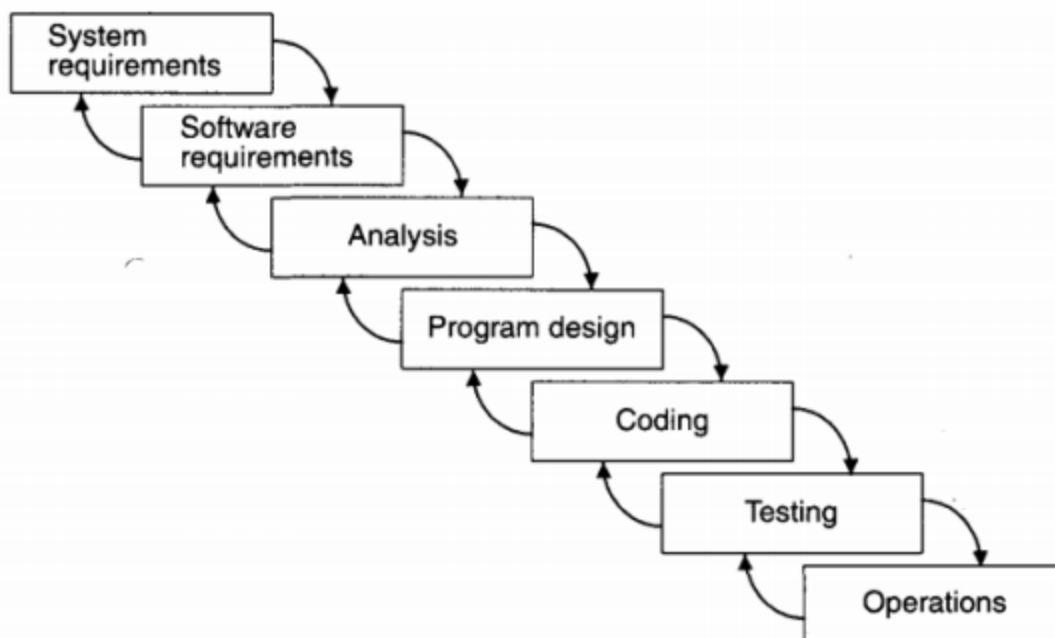
Three main primary points are

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.

Waterfall Model Part 2 : The large-scale system approach

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

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

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

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, and 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

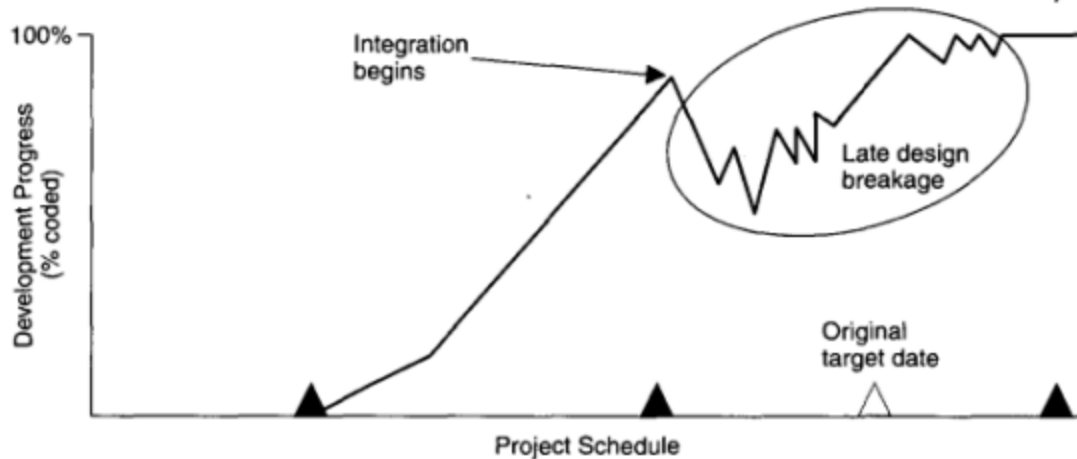
For a typical development project that used a waterfall model management process, below figure illustrates development progress versus time. Progress is defined as percent coded, that is, demonstrable in its target form.

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

Sequential activities: requirements — design — coding — integration — testing



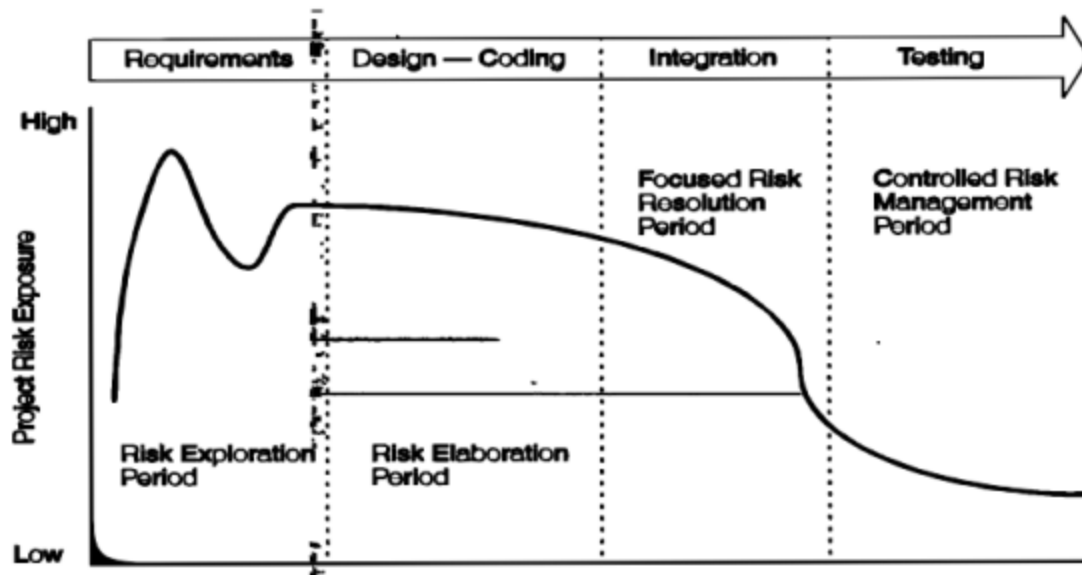
Progress profile of a conventional software project

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

Expenditures by activity for a conventional software project

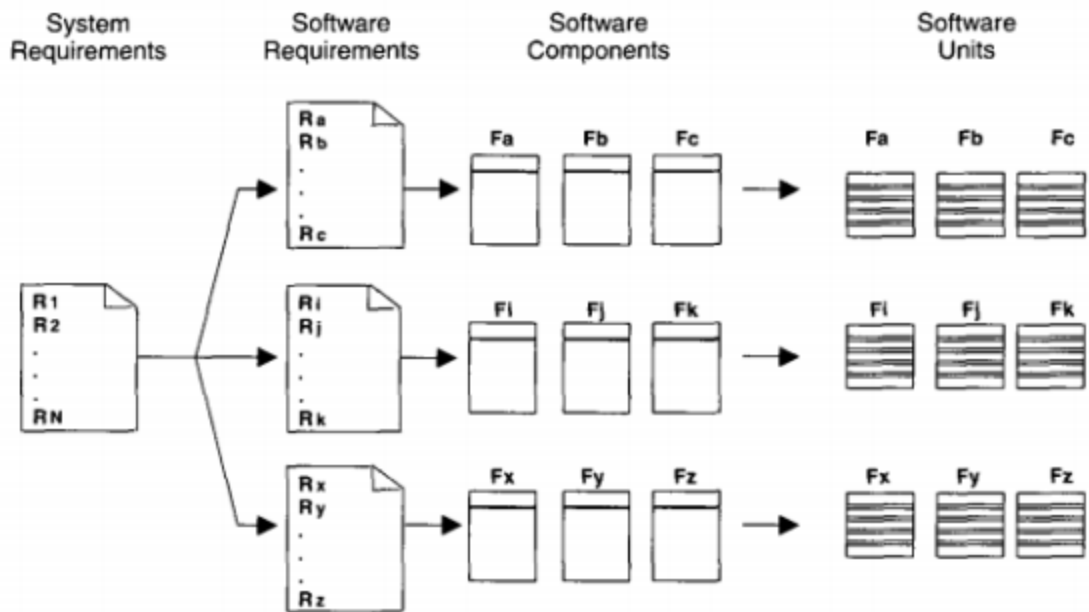
ACTIVITY	COST
Management	5%
Requirements	5%
Design	10%
Code and unit testing	30%
Integration and test	40%
Deployment	5%
Environment	5%
Total	100%

Late risk resolution A serious issue associated with the waterfall lifecycle was the lack of early risk resolution. Below Figure 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. Below Figure illustrates the result of requirements-driven approaches: a software structure that is organized around the requirements specification structure.



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. Below table summarizes the results of a typical design review.

<i>Results of conventional software project design reviews</i>	
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, and 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

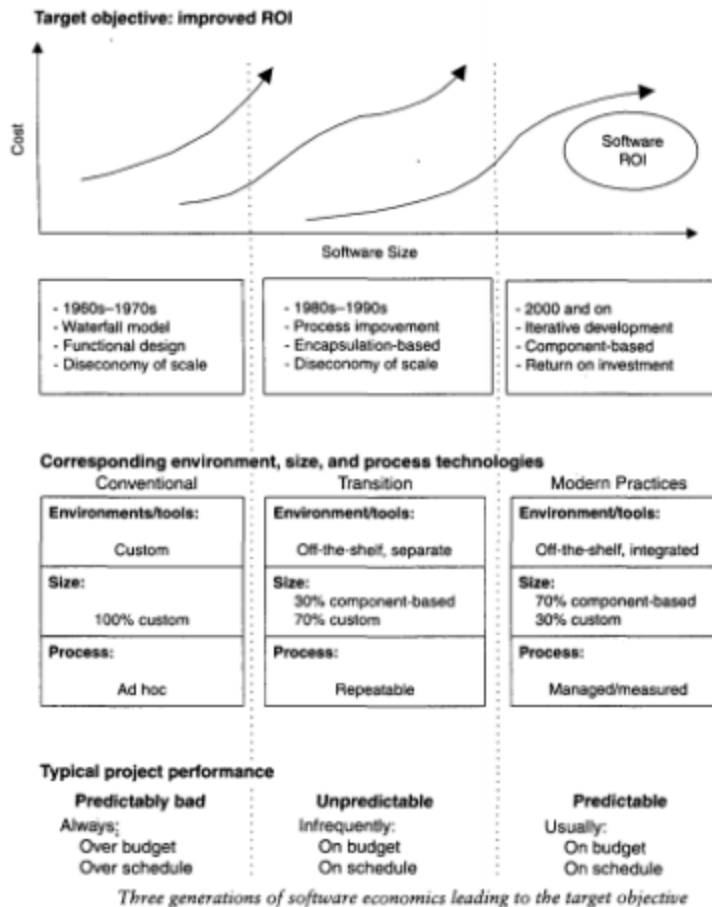
5. The required **quality** of the product, including its features, performance, reliability, and adaptability

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.

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

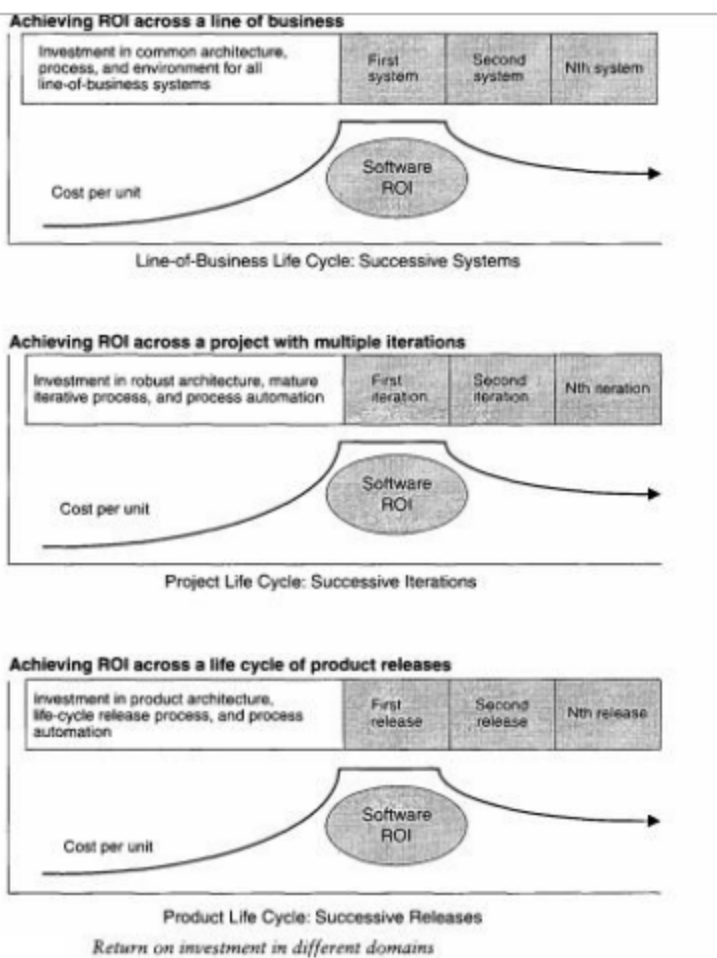


The three generations of software development are defined as follows:

- 1) Conventional:** 1960s and 1970s, craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable in that cost, schedule, and quality objectives were almost always underachieved.
- 2) Transition:** 1980s and 1990s, software engineering. Organizations used more-repeatable processes and off-the-shelf tools, and mostly (>70%) custom components built in higher level languages. Some of the components (<30%) were available as commercial products, including the operating system, database management system, networking, and graphical user interface.
- 3) Modern practices:** 2000 and later, software production. This book's philosophy is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70%) off-the-shelf components. Perhaps as few as 30% of the components need to be custom built. Technologies for environment automation, size reduction, and process improvement are not

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

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

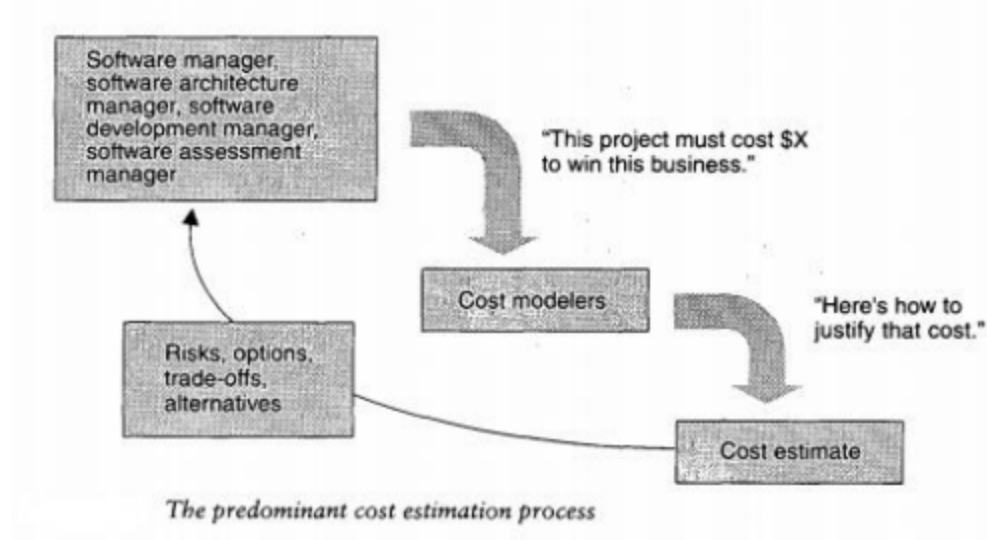
There have been many debates among developers and vendors of software cost estimation models and tools.

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), COCOMO is also one of the most open and well-documented cost estimation models. The general accuracy of conventional cost models (such as COCOMO) has been described as "within 20% of actuals, 70% of the time."

Most real-world use of cost models is bottom-up (substantiating a target cost) rather than top-down (estimating the "should" cost). The below Figure 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 the above Figure 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.

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.
4. Using better **environments**.
5. Trading off or backing off on **quality** thresholds.

These parameters are given in priority order for most software domains.

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

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

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.
- To reduce the software product size, the developers may follow **component based development model, object oriented technology, automatic code generators and so many tools.**
- The use of high level languages such as Java, .NET, Visual Basic for software development focused on few lines of human generated source code.

3.1.1 LANGUAGES

To estimate how much human generated code is reduced to decrease the software project size which is developed with high level languages and tools, there are two popular parameters.

One is **Universal Function Points (UFP)**, which consider external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries.

These are used to indicate relative program size to implement required functionality.

The second one is **SLOC** which is used to estimate the reduced size of the software project after a solution is formulated and an implemented language is known.

The selection of the language for the software project development is depending on what kind of the data that the application (project) be processed because each language has a domain of usage.

For example, Visual Basic is very expressive and powerful in building simple interactive applications, but it was not suitable for real time, embedded programming.

Some of these results are shown in the below Table.

Language expressiveness of some of today's popular languages

LANGUAGE	SLOC PER UFP
Assembly	320
C	128
FORTRAN 77	105
COBOL 85	91
Ada 83	71
C++	56
Ada 95	55
Java	55
Visual Basic	35

3.1.2 OBJECT-ORIENTED METHODS AND VISUAL MODELING

The object oriented technology reducing overall size of what needs to be developed. The following are three reasons for succeed of object oriented projects (proposed by BOOCH).

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.
2. The use of continuous integration creates opportunities to recognize risk early and make incremental corrections without destabilizing the entire development effort.
3. An object-oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire architecture.

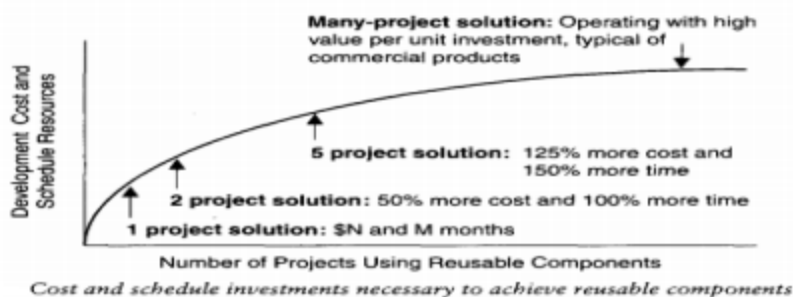
Booch also summarized five characteristics of a successful object-oriented projects.

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.

In general, things get reused for economic reasons. Therefore the key metric in identifying whether a component (or a class of components, or a commercial product) is truly reusable is to see whether some organization is making money on it.



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.

The below figure shows some of the advantages and disadvantages of using commercial components.

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. For software oriented organizations there are many processes and sub processes.

Generally the processes in organizations divided into three categories.

Metaprocess: which represents an organization's policies, procedures, and practices for pursuing the software business. Here we focus on organization economics, long-term strategies, and software ROI.

Macroprocess: it describes policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality.

Microprocess: it describes policies, procedures, and practices for achieving the goal of software by the development team. 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. The three levels of processes are overlapped at some point in the project development process, even though they have different objectives, audiences, metrics, concerns, and time scales as shown in the below Table.

Three levels of process and their attributes

ATTRIBUTES	METAPROCESS	MACROPROCESS	MICROPROCESS
Subject	Line of business	Project	Iteration
Objectives	Line-of-business profitability Competitiveness	Project profitability Risk management Project budget, schedule, quality	Resource management Risk resolution Milestone 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

3.3 IMPROVING TEAM EFFECTIVENESS

Managing the personnel has a great impact on the software project development process.

The COCOMO model describes that combination of personnel skills and experience has an impact on productivity as a factor of four.

At the same time it is difficult to measure the performance of the software team whether it is functioning correctly or wrongly. To form a good development team the managers follow the simple procedure “hiring of good people”.

The good software development team should have two characteristics, one is Balance and other one is coverage. Whenever a team is out of balance it is helpless.

It have the good combination of planners, designers, coders, testers, trainers and soon. It means a good team need covers strong individual persons at key positions. To maximize the team management we should follow the bellow points;

- 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.

In examining how to staff a software project Boehm offered the five staffing principles. They 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.

The software managers also have good leadership qualities to improve the team effectiveness. The following are some qualities that a software manager should have.

- 1. Hiring skills.** Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but at the same time difficult.
- 2. Customer-interface skill.** Avoiding adversarial relationships among stakeholders is a prerequisite for success.
- 3. Decision-making skill.** We know that a good leader run into problems and decision making.
- 4. Team-building skill.** The manager should establish trust, motivate progress, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.
- 5. Selling skill.** Successful managers must sell all stakeholders on decisions and priorities. Selling achievements against objectives.

3.4 IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

The tools and environments used in software development process have a linear effect on the productivity.

- ✓ Planning tools, requirements management tools, visual modeling tools, compilers, editors, debuggers, quality assurance analysis tools, test tools, and user interfaces provide crucial automation and support for evolving the software engineering objectives.
- ✓ In addition to above configuration management environments provide the foundation for executing and instrumenting the process.
- ✓ **Round-trip engineering** is a term used to describe key capability of environments that support iterative development.
- ✓ **Forward engineering** is the automation of one engineering artifact from another, more abstract representation.
For example, compilers and linkers have provided automated transition of source 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).
- ✓ **Round trip engineering** describes the environment support needed to change an artifact freely and have other artifacts automatically changed entire set of requirements, design, implementation and deployment artifacts.

3.5 ACHIEVING REQUIRED QUALITY

The following are the some key issues to improve the software quality.

- ✓ During the software development focuses on the key requirements and identifies the critical cases at the beginning of the development. The continuous monitoring of requirements makes the balance between the requirements evolution, design evolution, and plan evolution.
- ✓ Use the elegant metrics, indicators to measure the progress and quality of the software from its beginning of the life cycle to fully completed product
- ✓ Provide best environment where to integrate various processes to support configuration control, change management, rigorous design methods, document automation, and regression test automation.
- ✓ Use the advanced tools such as visual modeling techniques, higher level languages that support architectural control, abstraction, reliable programming, reuse, and self documentation.

- ✓ Providing integrated life-cycle environments that support early and continuous configuration control.
- ✓ Early and continuous insight into performance issues through demonstration-based evaluations

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

The following are some of key issues in project development life cycle which affects the quality of the software.

- ✓ Project beginning: The proposed design was declare to be low risk with adequate performance margin.
- ✓ Initial design review: Positive assessments of the design margin done mostly on paper. The actual application algorithms, databases are also fairly understood, but the operating system overhead, database management overhead, interprocesses and network communication overhead are secondary and misunderstood
- ✓ Mid-life -cycle design review: The assessment started in cut fashion at the middle, as early benchmarks and initial tests are correct.
- ✓ Integration and test: Serious performance problems are finding out then change the architecture. At this point the used infrastructure may be the scrap

3.6 PEER INSPECTIONS: A PRAGMATIC VIEW

Peer inspections are sometimes leads to overestimation of quality of the system. Generally peer reviews are important to compare the quality of the system against to the primary quality mechanisms and indicators.

The following are some of primary quality mechanisms to be followed during the development of the project.

- Transitioning engineering information from one artifact set to another, thereby assessing the consistency, feasibility, understandability, and technology constraints in engineering.
- 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 rigidity, consistency, completeness, and change control.
- Life-cycle testing for detailed insight into critical trade-offs, acceptance criteria, and requirements completeness.
- Change management metrics for objective insight into multiple-perspective change trends and convergence or divergence from quality and progress goals.

4. THE OLD WAY AND THE NEW

4.1 THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING: After years of software development experience, the software industry has learned many lessons and formulated many principles. The following are 30 major principles of 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 to learn user's 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 obvious solution.

- 5. Evaluate design alternatives:** After the requirements are approved, the developer must examine variety of architectures and algorithms. The architecture should relevance to the design 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:** Generally software development follows one notation throughout the development life cycle. But it is not optimal in all cases.
- 8. Minimize intellectual distance:** To minimize intellectual distance, the software 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.
- 10. Get it right before you make it faster:** It is 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. The right people with insufficient tools, languages, and process will succeed. Wrong people with appropriate tools, languages, and process will probably fail.
- 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:** one of the fact is the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.
- 17. The more they (customer) see, the more they need:** The more functionality if the developer provides to a user, the more functionality (or performance) the user wants.

- 18. Plan to throw one away:** One of the important success factors is whether or not the project is new. The brand new applications rarely work the first time.
- 19. Design for change:** The architecture, 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 the users more efficient.
- 22. Avoid tricks:** Some time the developers use tricks to develop the programs, those are functionally correct but in unclear way.
- 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:** use Coupling and cohesion are the best ways to measure software's natural maintainability and adaptability.
- 25. Use the McCabe complexity measure:** There are so many methods to measure the complexity of software, but use McCabe method which is simple one.
- 26. Don't test your own software:** The Software developers should never be the primary tester of their own software.
- 27. Analyze causes for errors:** First analyze the causes for errors when those are detected because it is cost effective to reduce the effect of an error by preventing it than to find and fix it.
- 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 exclusively by person-months makes little sense.
- 30. Expect excellence:** The team members will do much better if team leader have expectations for them.

4.2 THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT

The conventional software management principles are not suitable to the current trends followed in the industry. In this connection the modern software principles were evolved.

The following are ten (10) modern software principles

- 1. Base the process on an architecture-first approach:** To start the software development, the developer first thoroughly identifies the requirements then chooses the significant design decision.

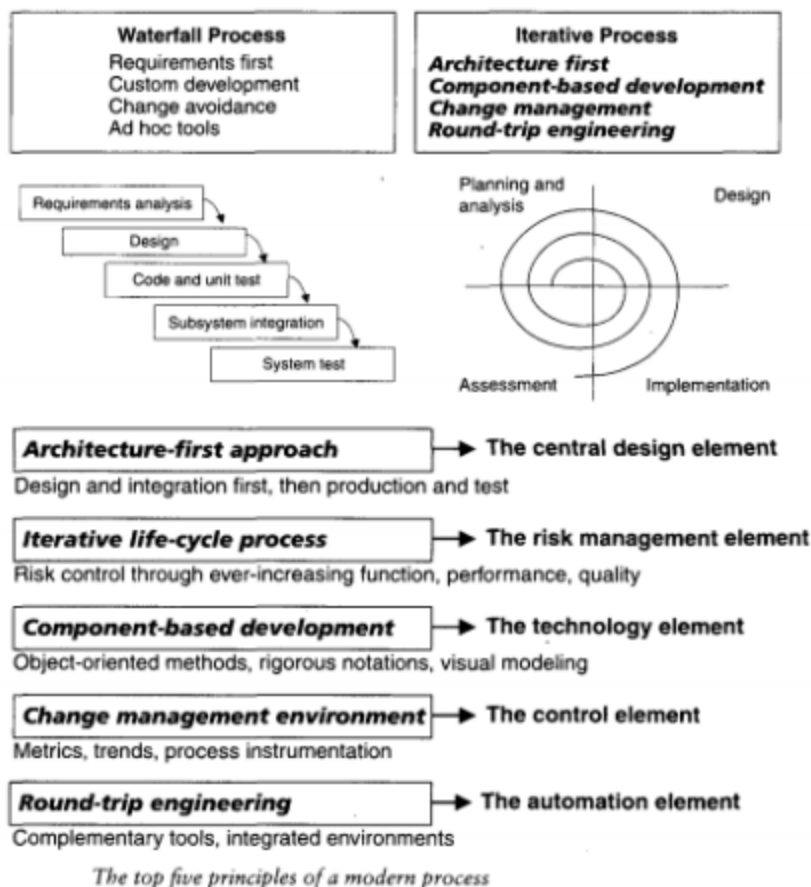
Before start the lifecycle (development) process check the selected resources are sufficient for complete software development.

2. Establish an iterative life-cycle process that confronts risk early: Today the organizations not follow to define the entire problem at once, design the entire solution, build the software, then test and deploy in sequence.

The organizations adopt follows iterative development approach which facilitates, problem understanding, effective solution, and an effective plan over several iterations. It also balances the stockholder's objectives. The main advantage of iterative approach is risks are identified early to avoid software scrap and rework.

3. Transition design methods to emphasize component-based development: The developers should transits from line-of-code mentality to component based mentality to reduce the amount of human generated code and custom development.

4. Establish a change management environment: The dynamics of iterative development, concurrent workflows require the controlled baselines.



5. **Enhance change freedom through tools that support round trip engineering (iterative):** The iterative development process supports 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:** Generally model based approach supports rich graphical and textual design notions. Visual modeling and formal machine process able language provides good measures than traditional design approach.
7. **Instrument the process for objective quality control and progress assessment:** Project development progress and the quality should be assessed in the middle of the development lifecycle.
8. **Use a demonstration-based approach to assess intermediate artifacts:** Transit from current development artifacts to executable one to early elimination of architectural defects.
9. **Plan intermediate releases in groups of usage situations with evolving levels of detail:** It is necessary that the software management process has continuous demonstrations within the operational context.
10. **Establish a configurable process that is economically scalable:** No single process is suitable for software development. A practical process frame work must be organizing a broad spectrum of applications. The software development process must integrate economy of scale, development automation, architecture patterns and components.

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.

To implement the modern techniques it is necessary that an initial version of the system be rapidly constructed early in the development process with an importance of addressing high-risk areas, stabilizing the basic structure, refining requirements, developing system on core architecture with number of reviews until achieving desired level of functionality, performance and robustness.

The iterative development gives importance to whole system rather than the individual parts. Risks are identified early through continuous integration and refinement of requirements, architecture, and plans.

We are also achieving some economical benefits with iterative approach, but to quantify the benefits is a difficult issue.

The following are some points represented by the COCOMO-II in the development of software projects by using the iterative approach.

- **Application precedentedness:** Domain experience is a critical factor in understanding how to plan and execute a software development project.
- **Process flexibility:** Today modern software has large solution space and many interrelated issues that there is a vital need for continuous changes. These changes natural in problem understanding, solution space, or in plans. So project development process must support an efficient change management.
- **Architecture risk resolution:** Architecture first development is a central theme in a successful iterative development process. The project team first develops and then stabilizes the architecture before developing all components that comprise entire application. This approach initiates integration activity early in the life cycle as the verification activity of the design process and products.
- **Team cohesion:** Successful teams are cohesive and cohesive teams are successful. The successful and cohesive teams share common objectives and priorities. Successful and cohesive teams avoid project confusion and entropy that may generated from stakeholders expectations
- **Software process maturity.** The software engineering institute's Capability Maturity Model (CMM) is a well accepted benchmark for software process assessment. The CMM is crucial important for avoiding software development risks and develops organization's software assets. Here we follow object quality control.

Important questions

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

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.

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

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:

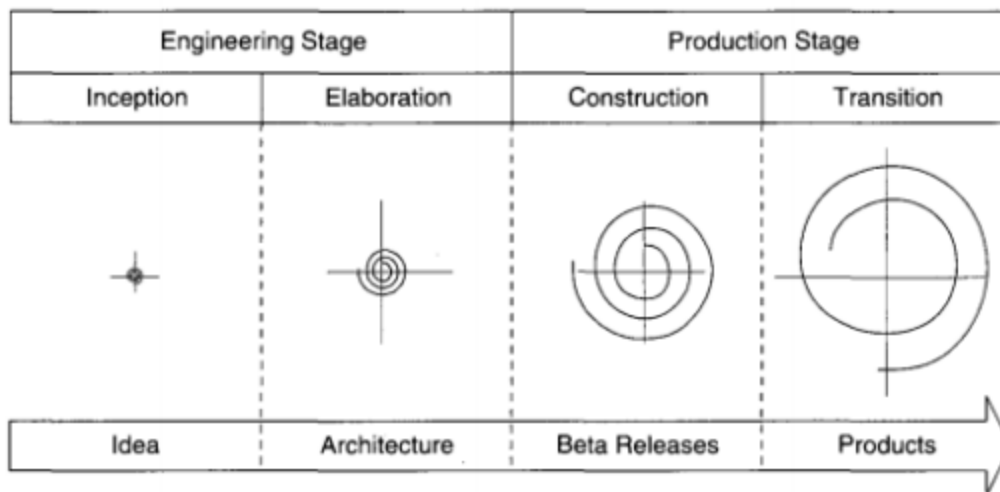
1. The **engineering stage**, driven by less predictable but smaller teams doing design and synthesis activities
2. The **production stage**, driven by more predictable but larger teams doing construction, test, and deployment activities

The below table summarizes the differences between these two stages.

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 the below Figure.

*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 ACTIVITIES

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

PRIMARY EVALUATION CRITERIA

- Do all stakeholders concur on the scope definition and cost and schedule estimates?
- Are requirements understood, as evidenced by the fidelity of the critical use cases?
- Are the cost and schedule estimates, priorities, risks, and development processes credible?
- Do the depth and breadth of an architecture prototype demonstrate the preceding criteria? (The Primary value of prototyping candidate architecture is to provide a vehicle for understanding the Scope and assessing the credibility of the development group in solving the particular technical problem.)
- Are actual resource expenditures versus planned expenditures acceptable?

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

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

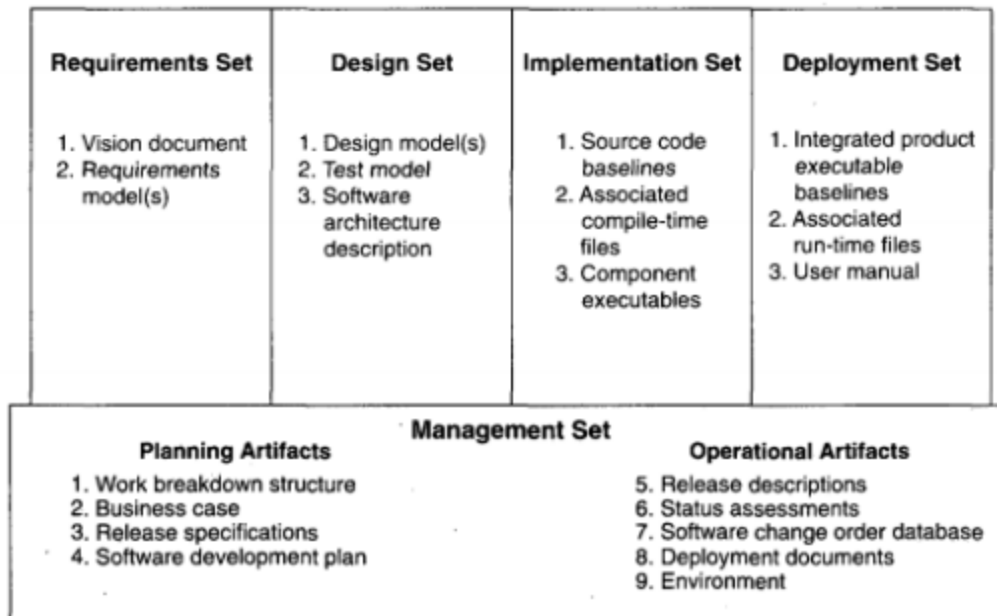
- Is the user satisfied?
- Are actual resource expenditures versus planned expenditures acceptable?

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-process able languages and associated files). The artifact sets are shown in the below Figure.



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.

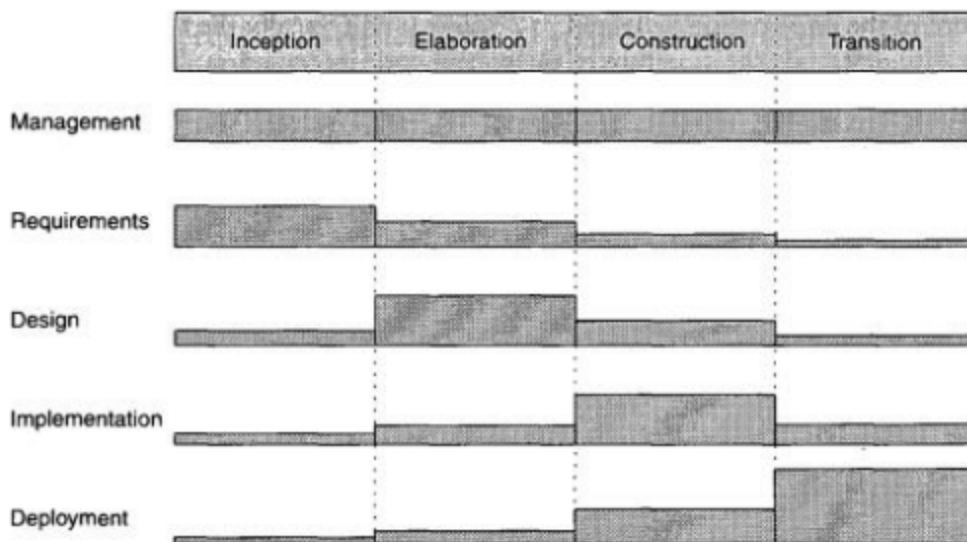
Deployment sets are evaluated, assessed, and measured through a combination of the following:

- 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.



Life-cycle focus on artifact sets

Implementation Set versus Deployment Set

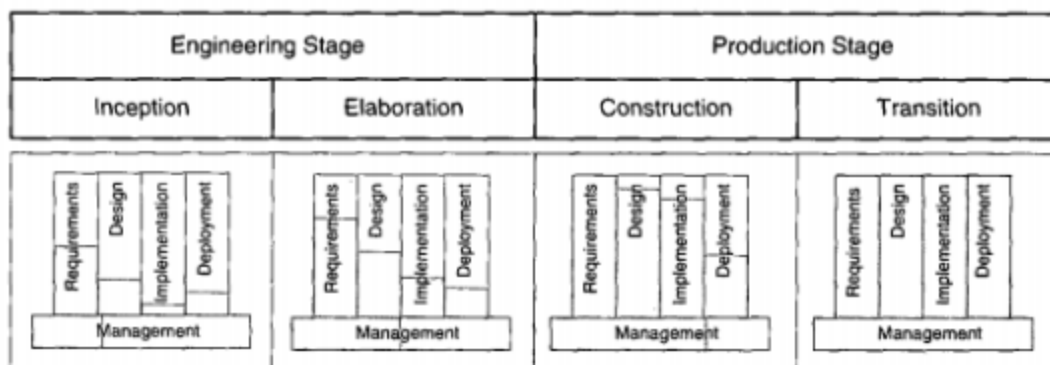
The separation of the implementation set (source code) from the deployment set (executable code) is important because there are very different concerns with each set. The structure of the information delivered to the user (and typically the test organization) is very different from

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, and 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 the below Figure.



Life-cycle evolution of the artifact sets

The **inception** phase focuses mainly on critical requirements usually with a secondary focus on an initial deployment view. During the **elaboration** phase, there is much greater depth in

requirements, much more breadth in the design set, and further work on implementation and deployment issues. The main focus of the **construction phase** is design and implementation. The main focus of the **transition phase** is on achieving consistency and completeness of the deployment set in the context of the other sets.

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 non deliverable 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, afterhours 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. The below Figure provides a default outline for a business case.

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

Typical business case outline

Software Development Plan

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

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

Typical software development plan outline

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. The below Figure provides a default outline for a release specification.

- | |
|--|
| <ul style="list-style-type: none"> I. Iteration content II. Measurable objectives <ul style="list-style-type: none"> A. Evaluation criteria B. Followthrough approach III. Demonstration plan <ul style="list-style-type: none"> A. Schedule of activities B. Team responsibilities IV. Operational scenarios (use cases demonstrated) <ul style="list-style-type: none"> A. Demonstration procedures B. Traceability to vision and business case |
|--|

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. The below Figure provides a default outline for a release description.

- | |
|--|
| <ul style="list-style-type: none"> I. Context <ul style="list-style-type: none"> A. Release baseline content B. Release metrics II. Release notes <ul style="list-style-type: none"> A. Release-specific constraints or limitations III. Assessment results <ul style="list-style-type: none"> A. Substantiation of passed evaluation criteria B. Follow-up plans for failed evaluation criteria C. Recommendations for next release IV. Outstanding issues <ul style="list-style-type: none"> A. Action items B. Post-mortem summary of lessons learned |
|--|

Typical release description outline

Status Assessments

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

Environment

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

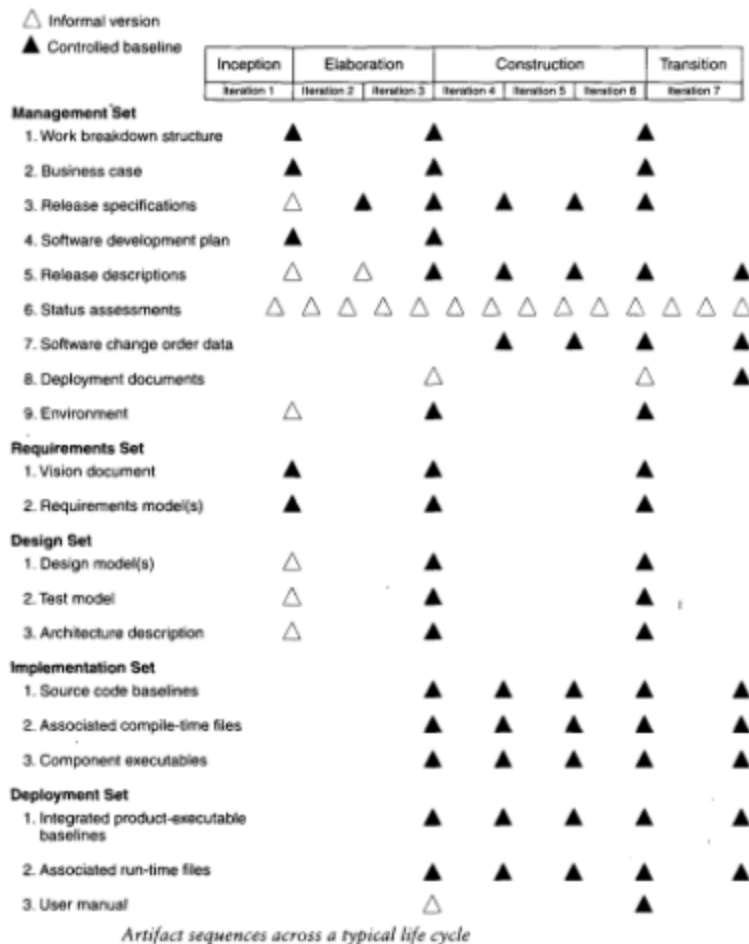
Deployment

A deployment document can take many forms. Depending on the project, it could include several document subsets for transitioning the product into operational status. In big contractual efforts in which the system 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. The below Figure identifies a typical sequence of artifacts across the life-cycle phases.



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. The below Figure 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)

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. The below Figure 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**

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.

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 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 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

The below Figure 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

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

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

Check Points of The process: Major Mile Stones, Minor Milestones, Periodic status assessments.

Iterative Process Planning: Work breakdown structures, planning guidelines, cost and schedule estimating, cost and schedule estimating, Iteration planning process, Pragmatic planning

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

Process Automation: Automation Building Blocks, the Project Environment

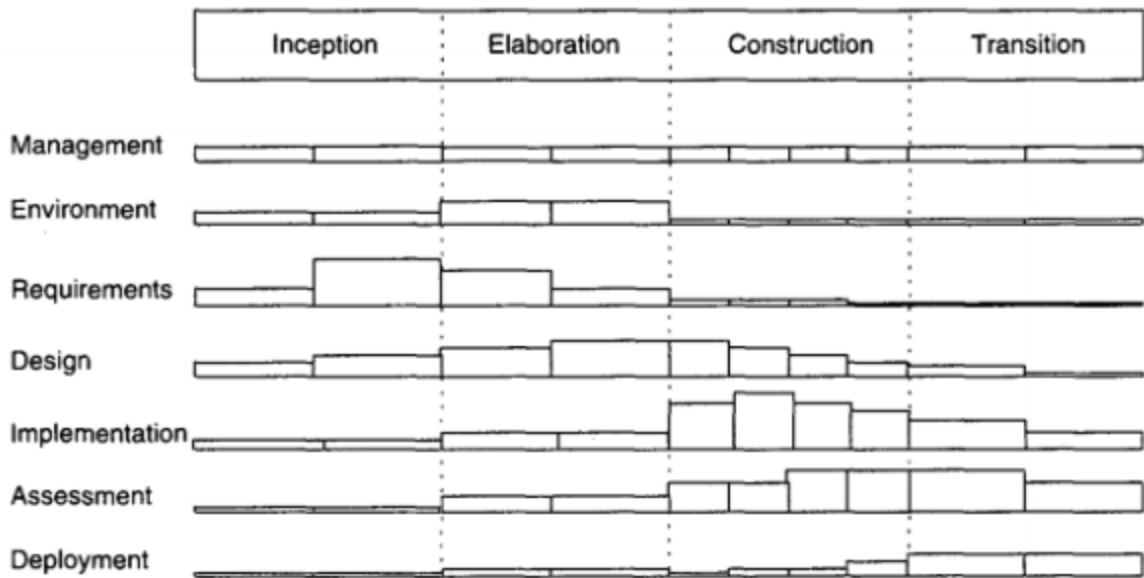
8. Workflow of the process

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

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



Activity levels across the life-cycle phases

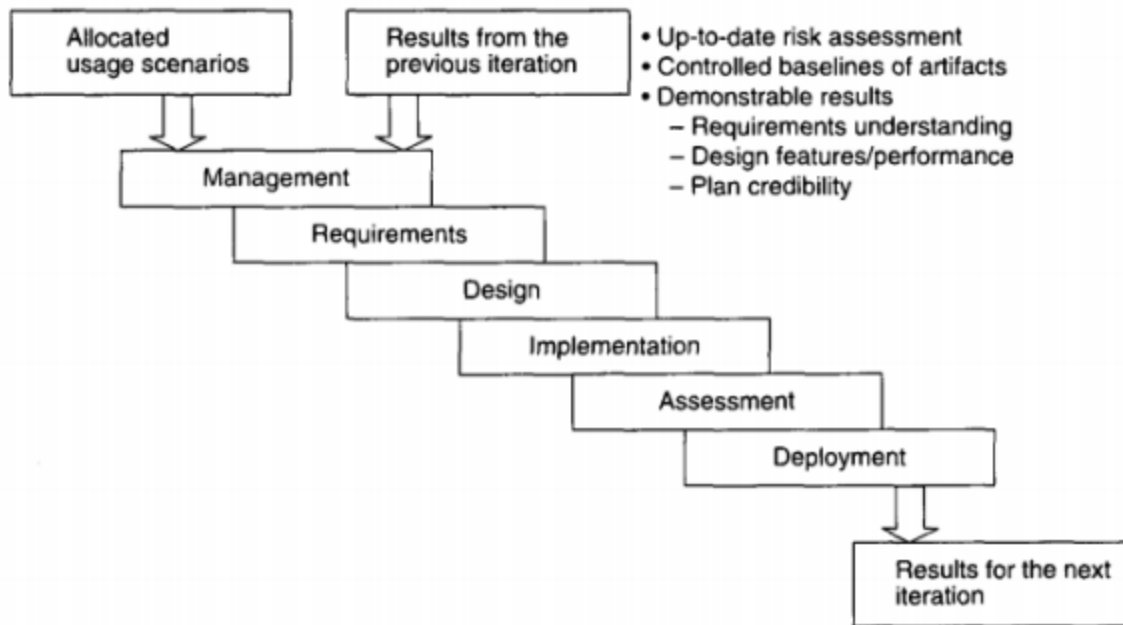
The below Table shows the allocation of artifacts and the emphasis of each workflow in each of the life-cycle phases of inception, elaboration, construction, and transition.

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 the below Figure generally includes the following sequence:



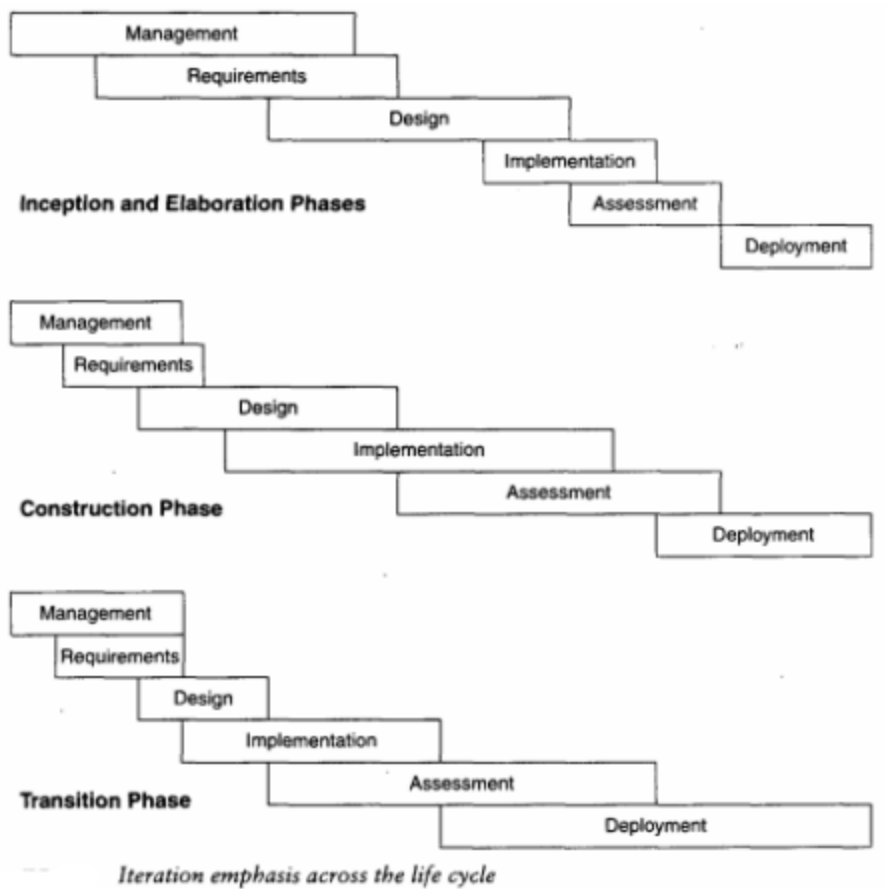
The workflow of an iteration

- Management: iteration planning to determine the content of the release and develop the detailed plan for the iteration; assignment of work packages, or tasks, to the development team
- Environment: evolving the software change order database to reflect all new baselines and changes to existing baselines for all product, test, and environment components
- Requirements: analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to be demonstrated at the end of this iteration and their evaluation criteria; updating any requirements set artifacts to reflect changes necessitated by results of this iteration's engineering activities
- Design: evolving the baseline architecture and the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate against the evaluation criteria allocated to this iteration; updating design set artifacts to reflect changes necessitated by the results of this iteration's engineering activities
- Implementation: developing or acquiring any new components, and enhancing or modifying any existing components, to demonstrate the evaluation criteria allocated to this iteration; integrating and testing all new and modified components with existing baselines (previous versions)

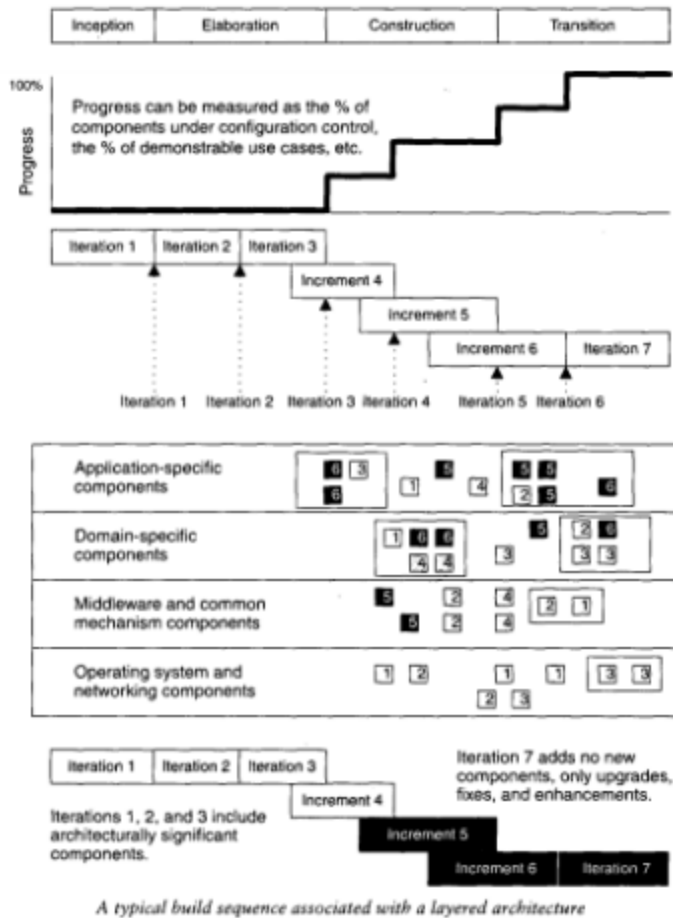
- **Assessment:** evaluating the results of the iteration, including compliance with the allocated evaluation criteria and the quality of the current baselines; identifying any rework required and determining whether it should be performed before deployment of this release or allocated to the next release; 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.

The below Figure 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.



The below Figure an example of a simple development life cycle, illustrates the differences between iterations and increments.



9. Checkpoints of the process

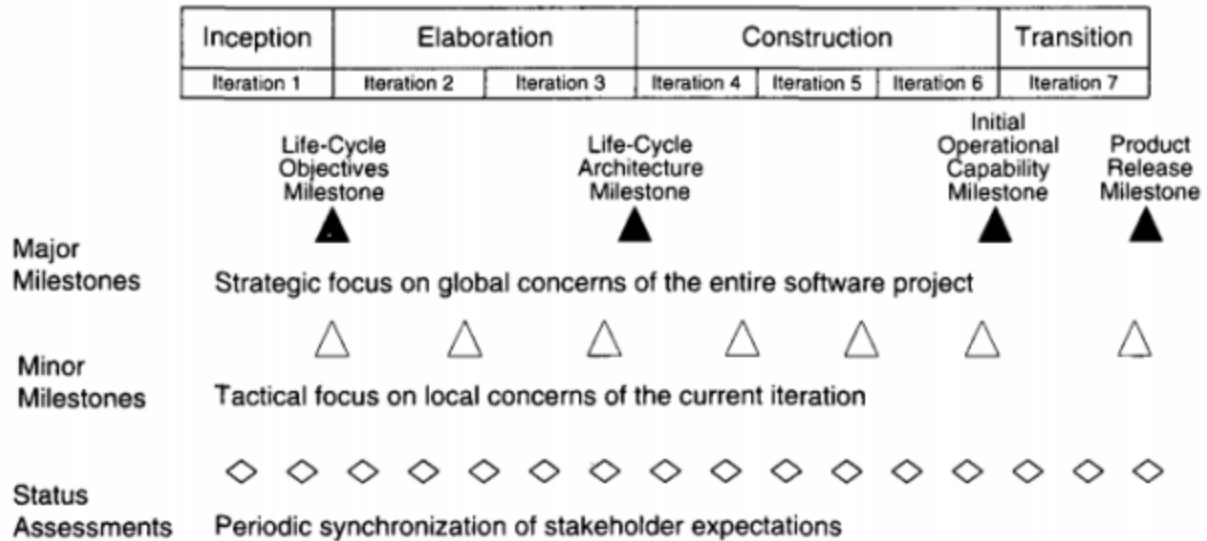
Three types of joint management reviews are conducted throughout the process:

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.

The below Figure illustrates a typical sequence of project checkpoints for a relatively large project.



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

The below Table summarizes the balance of information across the major milestones.

<i>The general status of plans, requirements, and products across the major milestones</i>			
MILESTONES	PLANS	UNDERSTANDING OF PROBLEM SPACE (REQUIREMENTS)	SOLUTION SPACE PROGRESS (SOFTWARE PRODUCT)
Life-cycle objectives milestone	Definition of stakeholder responsibilities	Baseline vision, including growth vectors, quality attributes, and priorities	Demonstration of at least one feasible architecture
	Low-fidelity life-cycle plan	Use case model	Make/buy/reuse trade-offs
	High-fidelity elaboration phase plan		Initial design model
Life-cycle architecture milestone	High-fidelity construction phase plan (bill of materials, labor allocation)	Stable vision and use case model	Stable design set
	Low-fidelity transition phase plan	Evaluation criteria for construction releases, initial operational capability	Make/buy/reuse decisions
		Draft user manual	Critical component prototypes
Initial operational capability milestone	High-fidelity transition phase plan	Acceptance criteria for product release	Stable implementation set
		Releasable user manual	Critical features and core capabilities
Product release milestone	Next-generation product plan	Final user manual	Objective insight into product qualities
			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 below Figure should have been reviewed by the time of the lifecycle Architecture milestone.

- | |
|---|
| <ul style="list-style-type: none"> I. Requirements <ul style="list-style-type: none"> A. Use case model B. Vision document (text, use cases) C. Evaluation criteria for elaboration (text, scenarios) II. Architecture <ul style="list-style-type: none"> 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) <ul style="list-style-type: none"> 1. Draft user manual III. Source and executable libraries <ul style="list-style-type: none"> A. Product components B. Test components C. Environment and tool components |
|---|

Engineering artifacts available at the life-cycle architecture milestone

The below Figure provides default agendas for this milestone.

Presentation Agenda

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

Demonstration Agenda

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

Default agendas for the life-cycle architecture milestone

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

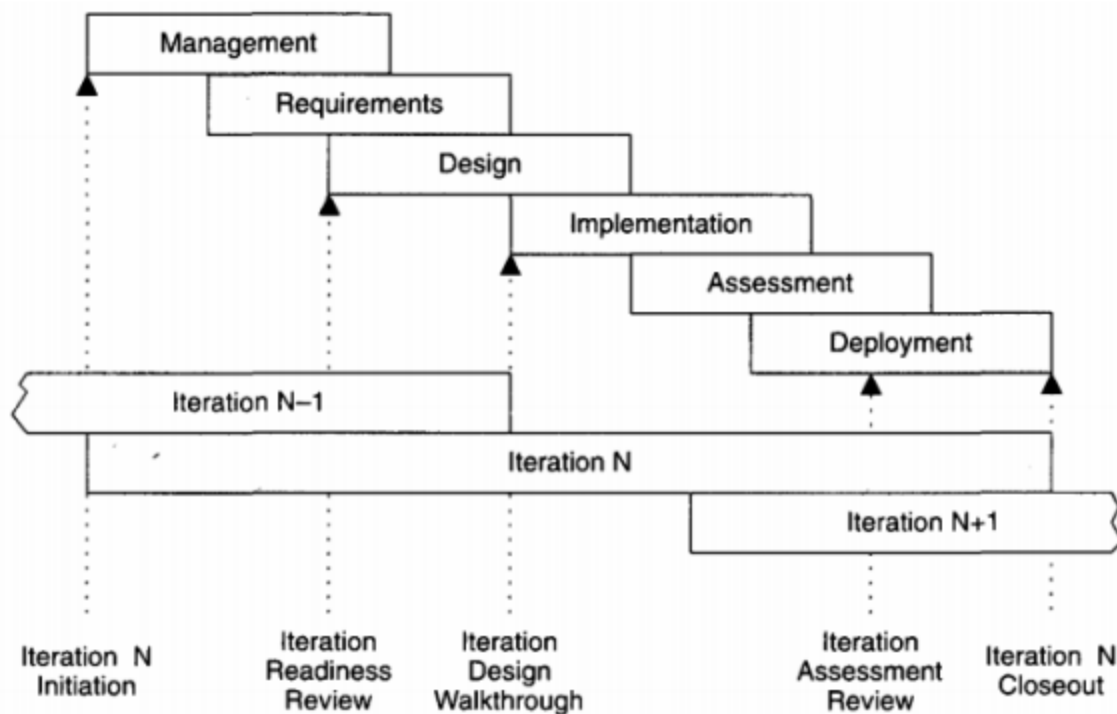
The product release milestone occurs at the end of the transition phase. The goal is to assess the completion of the software and its transition to the support organization, if any. The results of acceptance testing are reviewed, and all open issues are addressed. Software quality metrics are reviewed 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. The below Figure identifies the various minor milestones to be considered when a project is being planned.



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 the below Table.

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

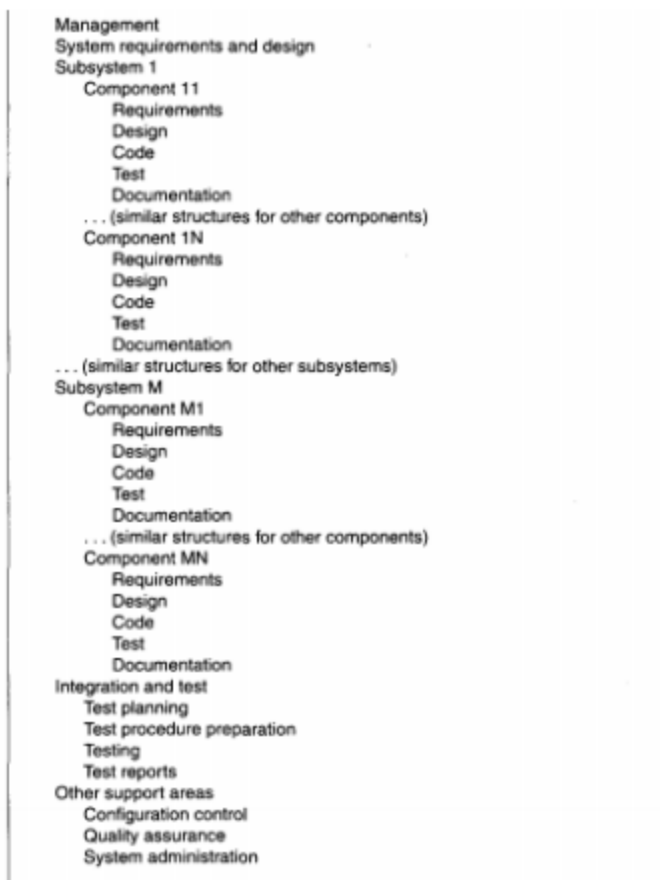
Many parameters can drive the decomposition of work into discrete tasks: product subsystems, components, functions, organizational units, life-cycle phases, even geographies. Most systems have a first-level decomposition by subsystem. Subsystems are then decomposed into their components, one of which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

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.

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 the below figure.

- A Management
 - AA Inception phase management
 - AAA Business case development
 - AAB Elaboration phase release specifications
 - AAC Elaboration phase WBS baselining
 - AAD Software development plan
 - AAE Inception phase project control and status assessments
 - AB Elaboration phase management
 - ABA Construction phase release specifications
 - ABB Construction phase WBS baselining
 - ABC Elaboration phase project control and status assessments
 - AC Construction phase management
 - ACA Deployment phase planning
 - ACB Deployment phase WBS baselining
 - ACC Construction phase project control and status assessments
 - AD Transition phase management
 - ADA Next generation planning
 - ADB Transition phase project control and status assessments
- B Environment
 - BA Inception phase environment specification
 - BB Elaboration phase environment baselining
 - BBA Development environment installation and administration
 - BBB Development environment integration and custom toolsmithing
 - BBC SCO database formulation
 - BC Construction phase environment maintenance
 - BCA Development environment installation and administration
 - BCB SCO database maintenance
 - BD Transition phase environment maintenance
 - BDA Development environment maintenance and administration
 - BDB SCO database maintenance
 - BDC Maintenance environment packaging and transition
- C Requirements
 - CA Inception phase requirements development
 - CAA Vision specification
 - CAB Use case modeling
 - CB Elaboration phase requirements baselining
 - CBA Vision baselining
 - CBB Use case model baselining
 - CC Construction phase requirements maintenance
 - CD Transition phase requirements maintenance

D	Design
DA	Inception phase architecture prototyping
DB	Elaboration phase architecture baselining
DBA	Architecture design modeling
DBB	Design demonstration planning and conduct
DBC	Software architecture description
DC	Construction phase design modeling
DCA	Architecture design model maintenance
DCB	Component design modeling
DD	Transition phase design maintenance
E	Implementation
EA	Inception phase component prototyping
EB	Elaboration phase component implementation
EBA	Critical component coding demonstration integration
EC	Construction phase component implementation
ECA	Initial release(s) component coding and stand-alone testing
ECB	Alpha release component coding and stand-alone testing
ECC	Beta release component coding and stand-alone testing
ECD	Component maintenance
ED	Transition phase component maintenance
F	Assessment
FA	Inception phase assessment planning
FB	Elaboration phase assessment
FBA	Test modeling
FBB	Architecture test scenario implementation
FBC	Demonstration assessment and release descriptions
FC	Construction phase assessment
FCA	Initial release assessment and release description
FCB	Alpha release assessment and release description
FCC	Beta release assessment and release description
FD	Transition phase assessment
FDA	Product release assessment and release descriptions
G	Deployment
GA	Inception phase deployment planning
GB	Elaboration phase deployment planning
GC	Construction phase deployment
GCA	User manual baselining
GD	Transition phase deployment
GDA	Product transition to user

Default work breakdown structure

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.

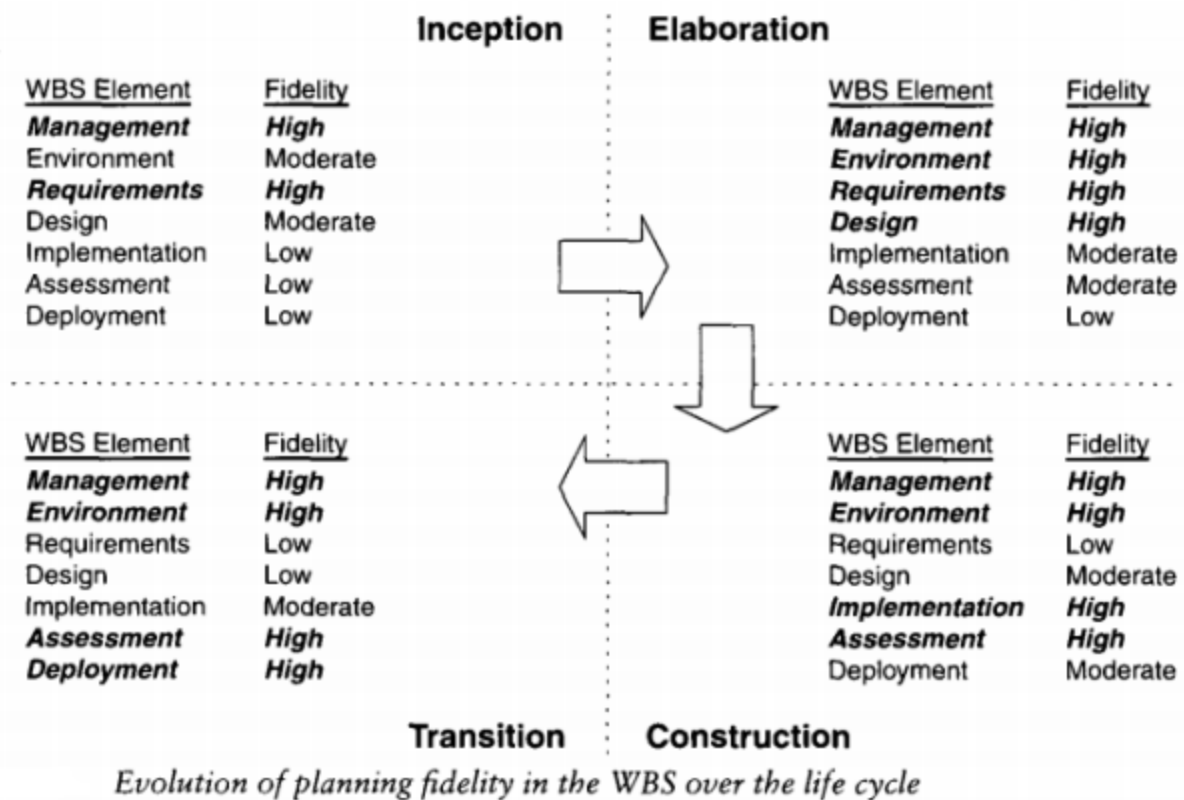
- Scale. Larger projects will have more levels and substructures.
- Organizational structure. Projects that include subcontractors or span multiple organizational entities may introduce constraints that necessitate different WBS allocations.

- 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.

The below Figure 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).

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

WBS budgeting defaults

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

The second guideline, detailed in the below Table, prescribes the allocation of effort and schedule across the lifecycle phases.

Default distributions of effort and schedule by phase

DOMAIN	INCEPTION	ELABORATION	CONSTRUCTION	TRANSITION
Effort	5%	20%	65%	10%
Schedule	10%	30%	50%	10%

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, and then decomposes these elements into lower level budgets and intermediate milestones. From this perspective, the following planning sequence would occur:

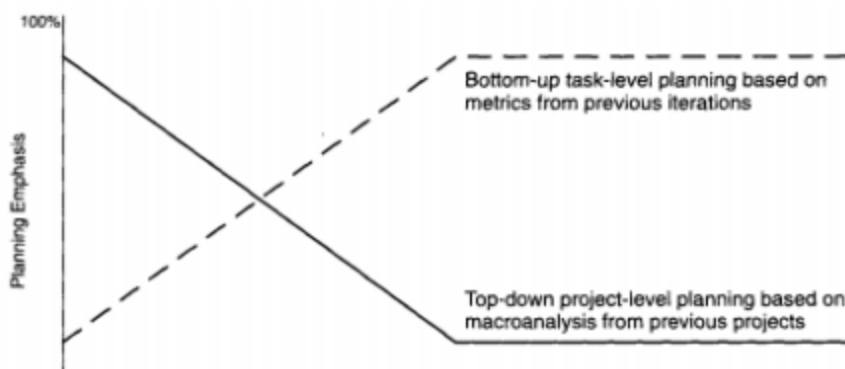
1. The software project manager (and others) develops a characterization of the overall size, process, environment, people, and quality required for the project.
2. A macro-level estimate of the total effort and schedule is developed using a software cost estimation model.
3. The software project manager partitions the estimate for the effort into top-level WBS using guidelines.
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, and then sum all these elements into the higher level budgets and intermediate milestones. This approach tends to define and populate the WBS from the lowest levels upward. From this perspective, the following planning sequence would occur:

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

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

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



Engineering Stage		Production Stage	
Inception	Elaboration	Construction	Transition
Feasibility iterations	Architecture iterations	Usable iterations	Product releases

Engineering stage planning emphasis:

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

Production stage planning emphasis:

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

Planning balance throughout the life cycle

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.

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.

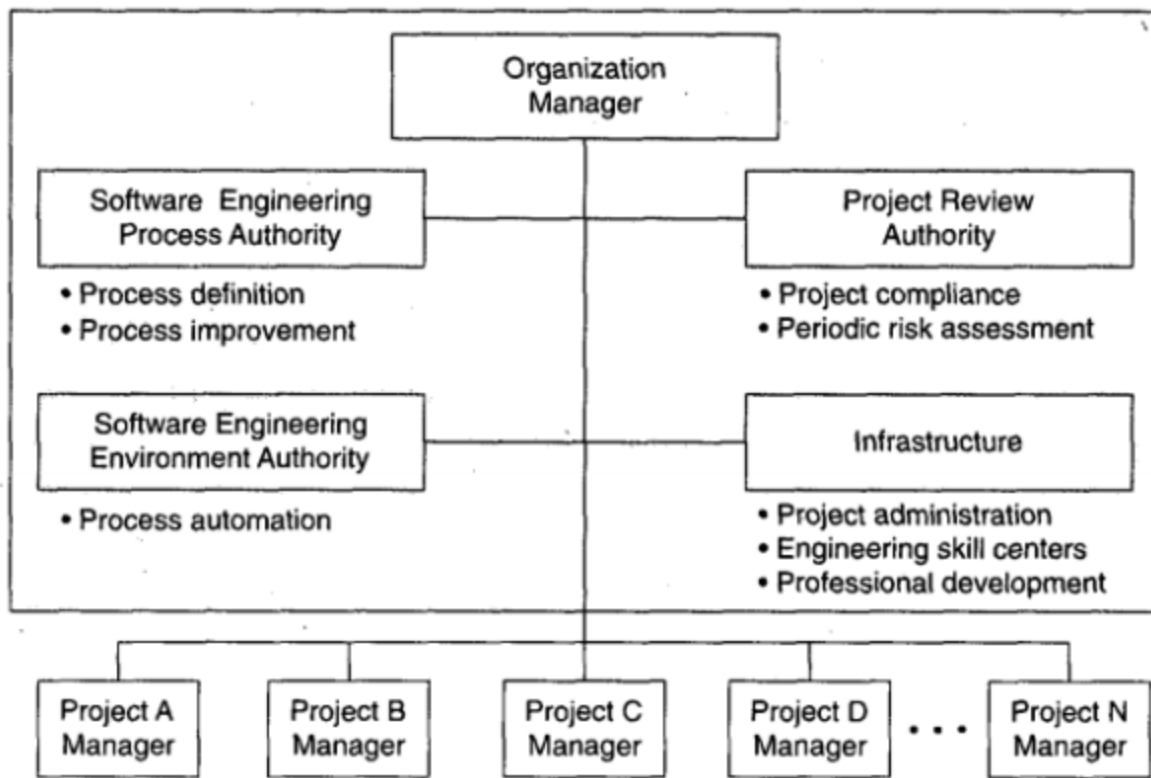
Software lines of business are motivated by return of investment (ROI), new business discriminators, market diversification & profitability.

Project teams are motivated by the cost, Schedule & quality of specific deliverables

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.



Default roles in a software line-of-business organization

Software Engineering Process Authority (SEPA)

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

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

Project Review Authority (PRA)

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

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

Software Engineering Environment Authority (SEEA)

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

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

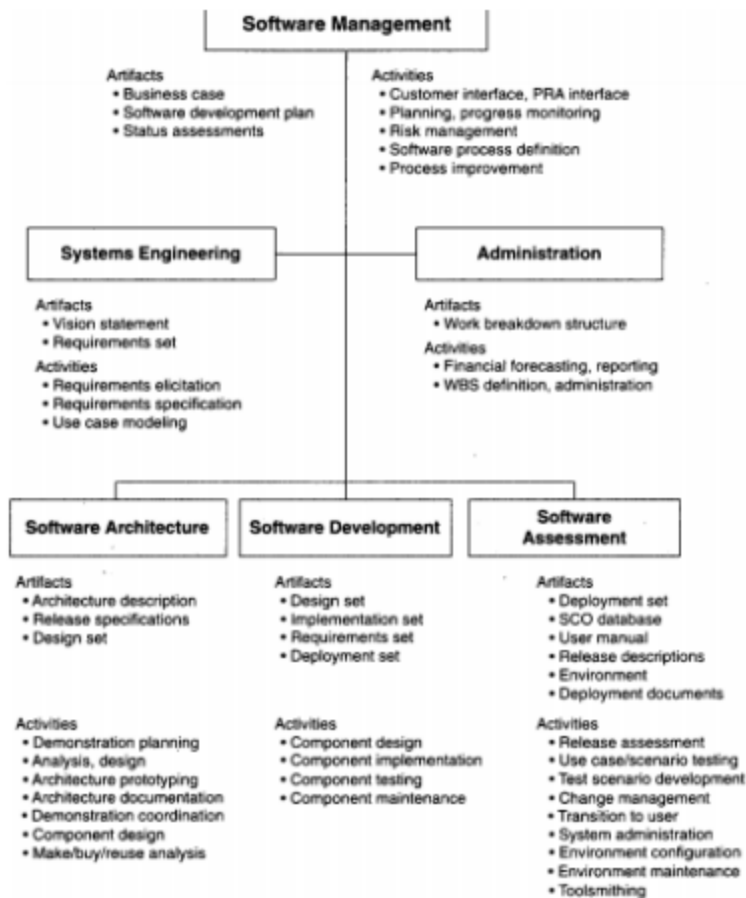
Infrastructure

An organization's infrastructure provides human resources support, project independent research & development, & other capital software engineering assets. The typical components of the organizational infrastructure are as follows

- **Project Administration:** Time accounting system; contracts, pricing, terms and condition; corporate information system integration.
- **Engineering Skill Centers:** Custom tools repository and maintenance, bid and proposal support, independent research and development.
- **Professional Development:** Internal training boot camp, personnel recruiting, personnel skills database maintenance, literature and assets library, technical publications.

PROJECT ORGANIZATIONS

The below figure shows a default project organization and maps project-level roles and responsibilities.



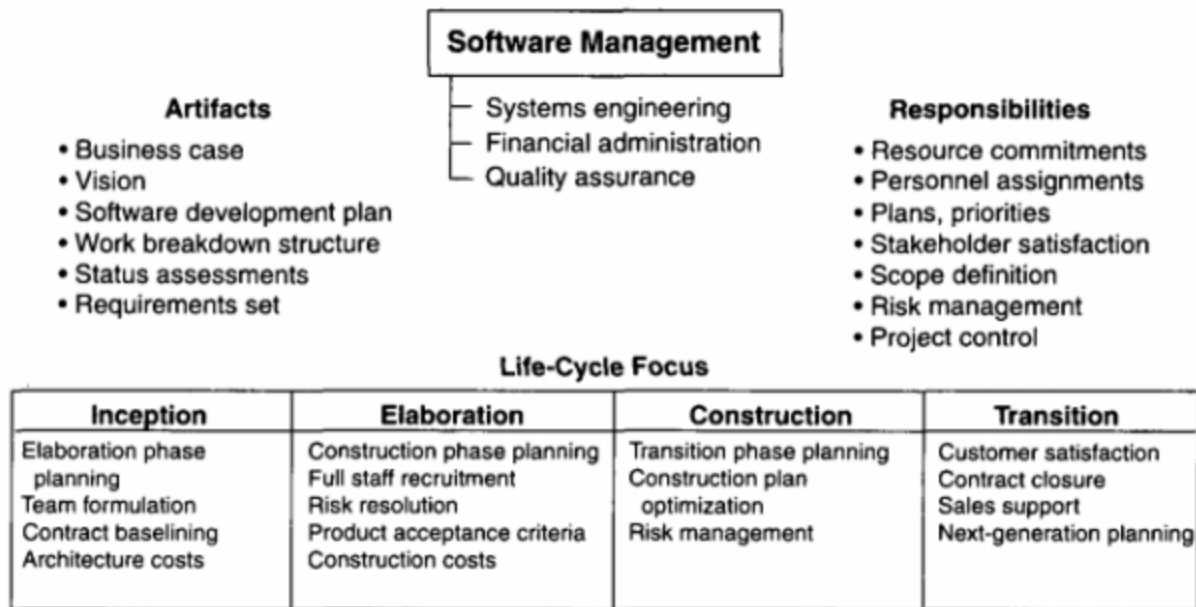
Default project organization and responsibilities

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.

Software Management Team: Most projects are over constrained. Schedules, cost, functionality and quality expectations are highly inter related and require continuous negotiation among multiple stake holders who have different goals. The software management team carries the burden

of delivering with condition to all stake holders. The software management team takes ownership of all aspects of quality.



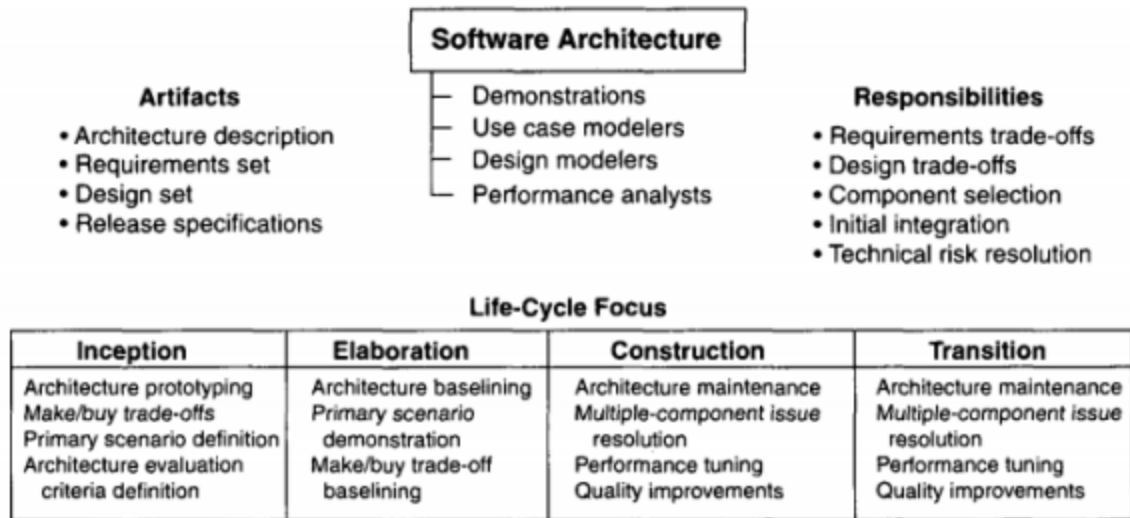
Software management team activities

Software Architecture Team:

The software architecture team is responsible for the architecture. This responsibility encompasses the engineering necessary to specify a complete bill of materials for the software and the engineering necessary to make significant make/ buy trade-offs so that all custom components are elaborated to the extent that construction/assembly costs are highly predictable.

In most projects, the inception and elaboration phases will be dominated by two distinct teams: the software management team and the software architecture team. To succeed, the architecture must include a fairly broad level of expertise, including the following:

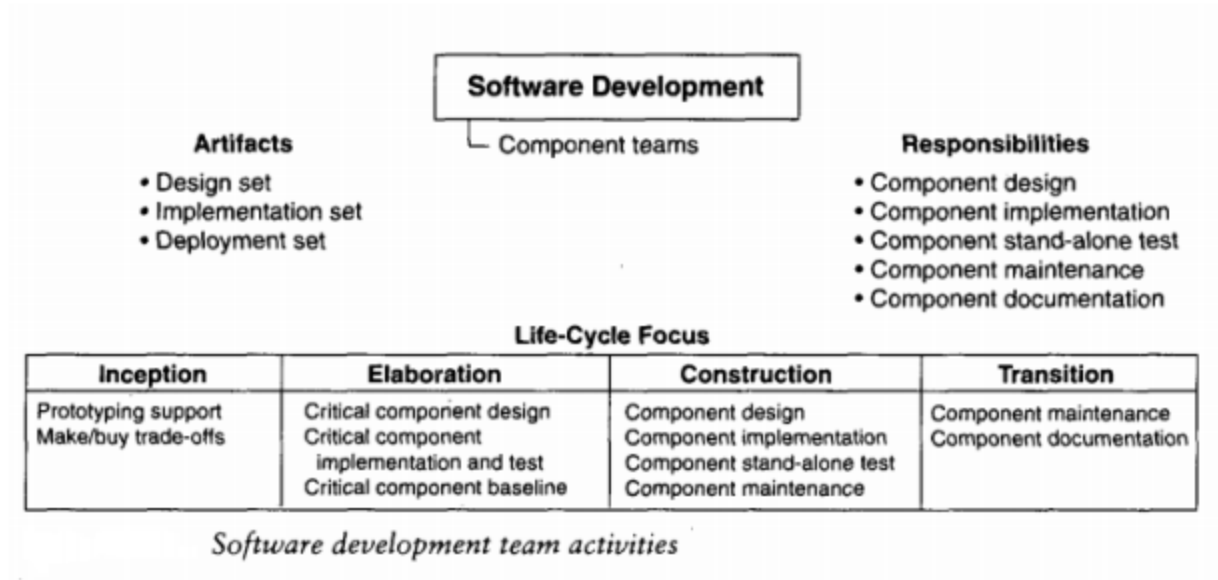
- Domain experience to produce an acceptable design views (architecturally significant elements of the design model) and use case view (architecturally significant elements of the use case model).
- Software technology experience to produce an acceptable process view (concurrency and control thread relationships among the design, component, and deployment models), component view (structure of the implementation set), and deployment view (structure of the deployment set).



Software architecture team activities

Software development team: The software development team is the most application specific group. In general, the software development team comprises several sub teams dedicated to groups of components that require a common skill set. The typical skill set include the following:

- Commercial component: Specialists with detail knowledge of commercial components central to a system's architecture
- Database: specialists with experience in the organization, storage, and retrieval of data
- Graphical user interfaces: specialists with experience in the display organization, data presentation, and user interaction.
- Operating systems and networking: specialist with experience in the execution of multiple software objects on a network of hardware resources.
- Domain applications: specialists with experience in the algorithms, application processing.



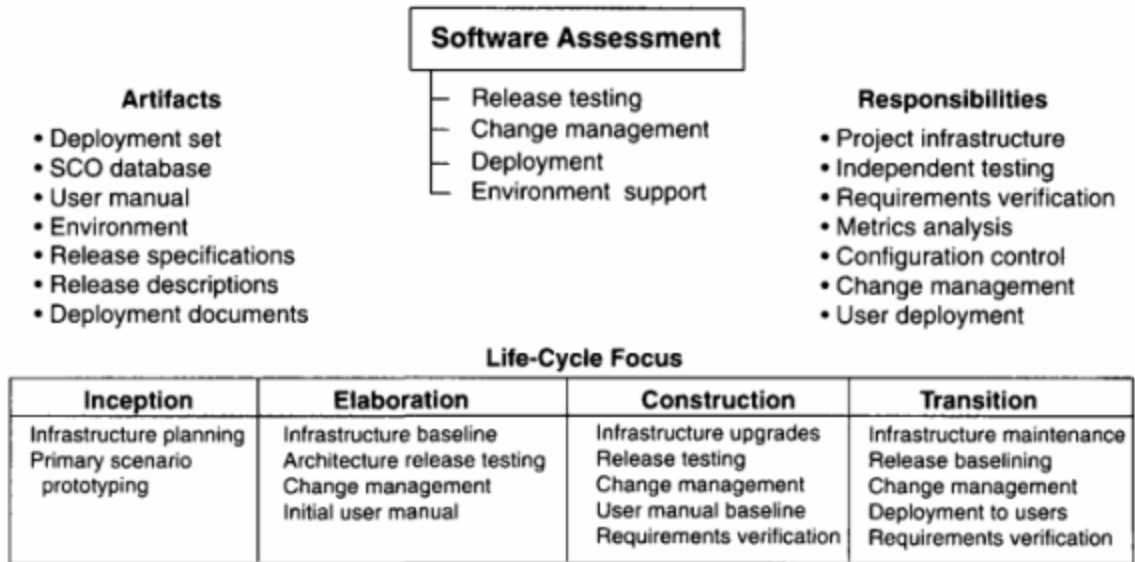
The software development team is responsible for the quality of individual components, including all component development, testing, and maintenance. Components tests should be built as self-documented.

Software Assessment Team:

There are two reasons for using an independent team for software assessment. It has to do with ensuring an independent quality perspective. A more important reason for using an independent test team is to exploit the concurrency of activities.

A modern process should employ use-case-oriented or capability –based testing (which may span many components). Organized as a sequence of builds and mechanized via two artifacts.

- Release specification (the plan and evaluation criteria for a release)
- Release description(the results of a release)



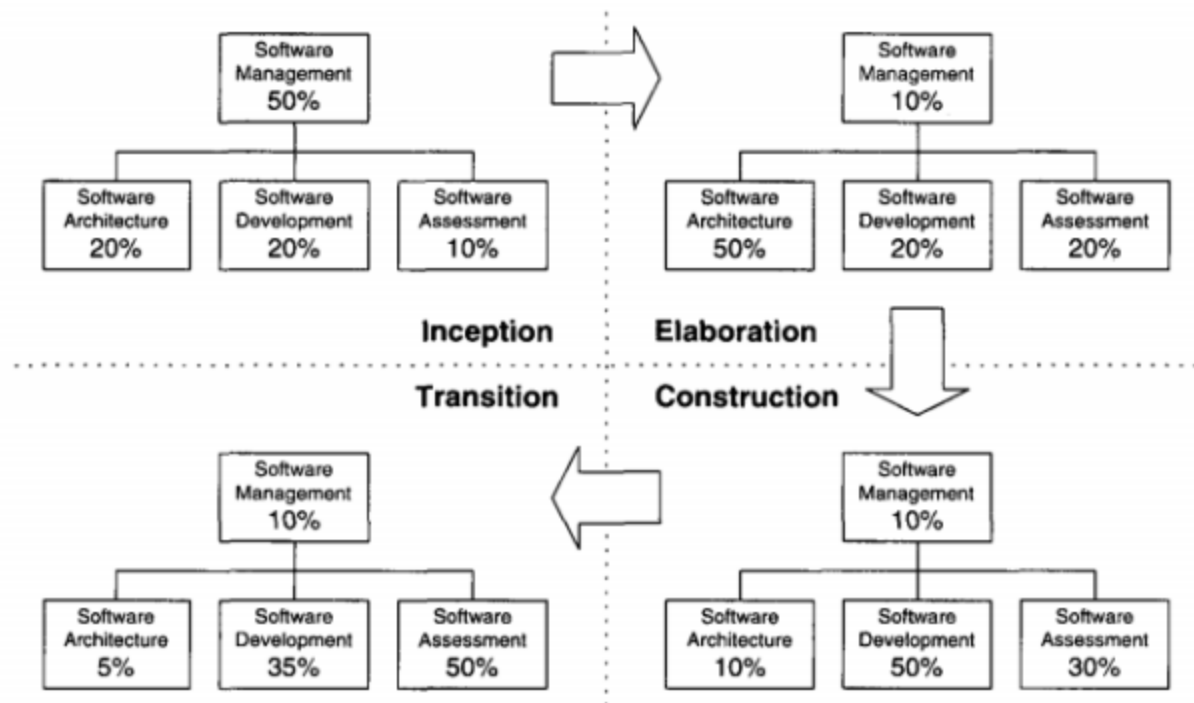
Software assessment team activities

EVOLUTION OF ORGANIZATIONS

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

A different set of activities is emphasized in each phase, as follows:

Inception team: an organization focused on planning, with enough support from the other teams to ensure that the plans represent a consensus of all perspectives.



Software project team evolution over the life cycle

- **Elaboration team:** an architecture focused organization in which the driving forces of the project reside in the software architecture team and are supported by the software development software assessment teams has necessary to achieve a stable architecture baseline
- **Construction team:** a fairly balanced organization in which most of the activity resides in the software development and software assessment teams
- **Transition team:** a customer focus organization in which usage feedback drives the deployment activities.

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.

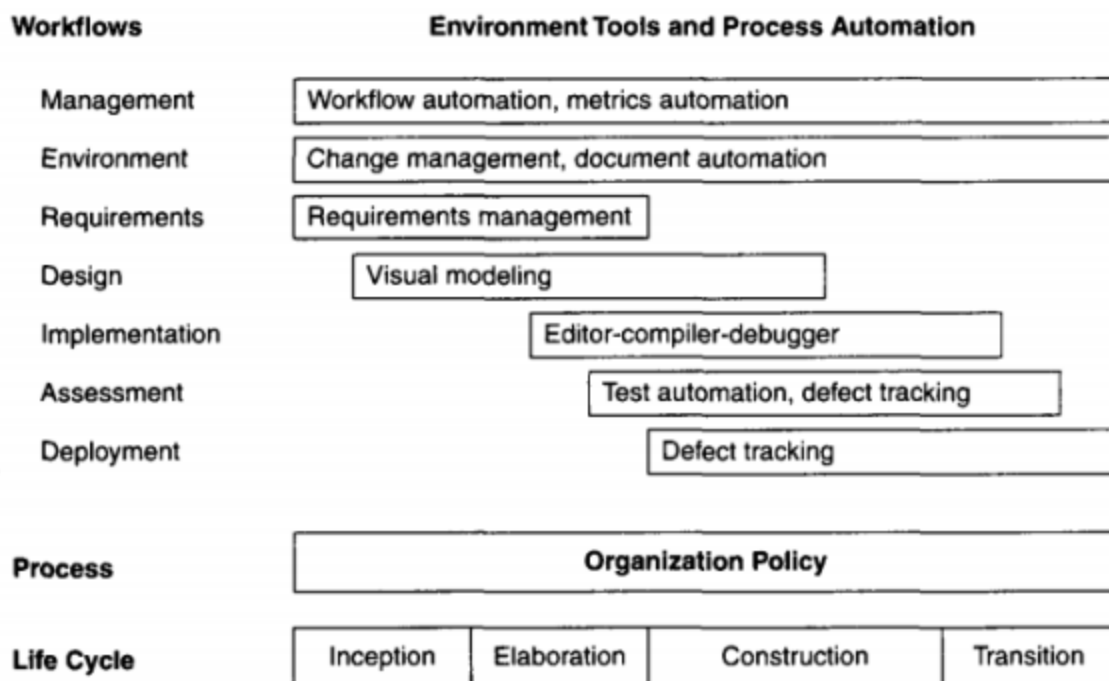
Metaproces(Line of business): The automation support for this level is called an infrastructure.

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

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

TOOLS: AUTOMATION BUILDING CLOCKS:

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



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.

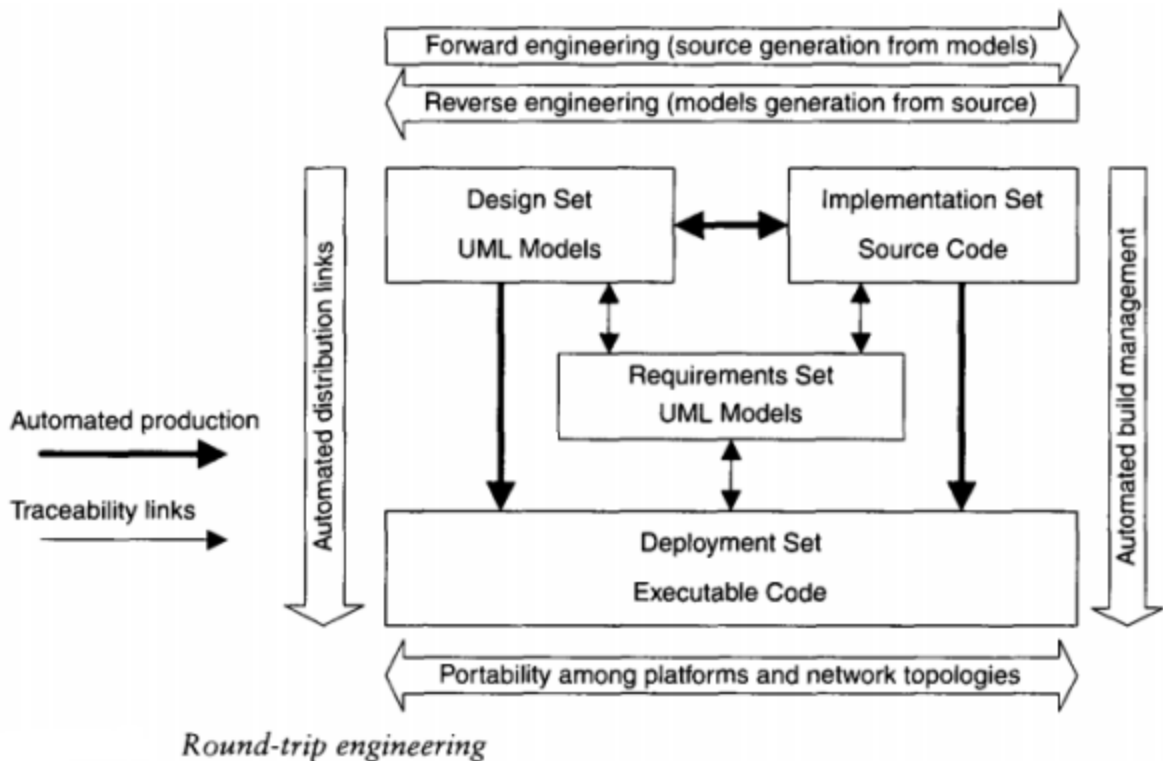
Round Trip Environment

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.



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: _____

The primitive components of a software change order

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.

III Configuration Control Board (CCB)

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

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

delivery system?

Infrastructure

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

I Organization Policy:

A Policy captures the standards for project software development processes. the organization policy is usually packaged as a handbook that defines the life cycles & the process primitives such as

- ✓ Major milestones

- ✓ Intermediate Artifacts
- ✓ Engineering repositories
- ✓ Metrics
- ✓ Roles & Responsibilities

- I. Process-primitive definitions**
 - A. Life-cycle phases (inception, elaboration, construction, transition)
 - B. Checkpoints (major milestones, minor milestones, status assessments)
 - C. Artifacts (requirements, design, implementation, deployment, management sets)
 - D. Roles and responsibilities (PRA, SEPA, SEEA, project teams)
- II. Organizational software policies**
 - A. Work breakdown structure
 - B. Software development plan
 - C. Baseline change management
 - D. Software metrics
 - E. Development environment
 - F. Evaluation criteria and acceptance criteria
 - G. Risk management
 - H. Testing and assessment
- III. Waiver policy**
- IV. Appendixes**
 - A. Current process assessment
 - B. Software process improvement plan

Organization policy outline

II Organization Environment

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

Stakeholder Environment

Many large scale projects include people in external organizations that represent other stakeholders participating in the development process they might include

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

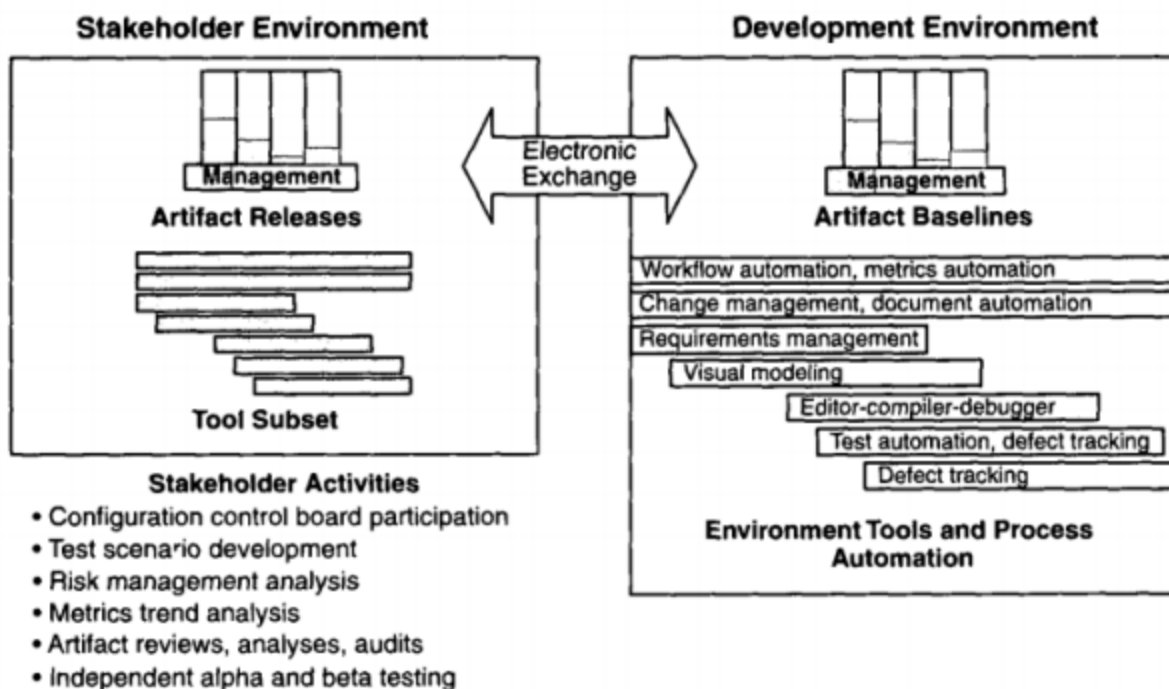
These stakeholder representatives also need to access to development resources so that they can contribute value to overall effort. These stakeholders will be access through on-line

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

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

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

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



Extending environments into stakeholder domains

UNIT V

13. Project Control and Process instrumentation

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

Need for Software Metrics:

- Software metrics are needed for calculating the cost and schedule of a software product with great accuracy.
- Software metrics are required for making an accurate estimation of the progress.
- The metrics are also required for understanding the quality of the software product.

1.1 INDICATORS:

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

Two types of indicators are:

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

1.1.1 Management Indicators

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

1.1.2 Quality Indicators

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

1.2 SEVEN CORE METRICS OF SOFTWARE PROJECT

Many different metrics may be of value in managing a modern process.

There are seven core metrics that should be used on all software projects. Three are management indicators and four are quality indicators.

Seven core metrics related to project control:

Management Indicators

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

Quality Indicators

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

1.2.1 MANAGEMENT INDICATORS:

1.2.1.1 Work and progress

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

The default perspectives of this metric are:

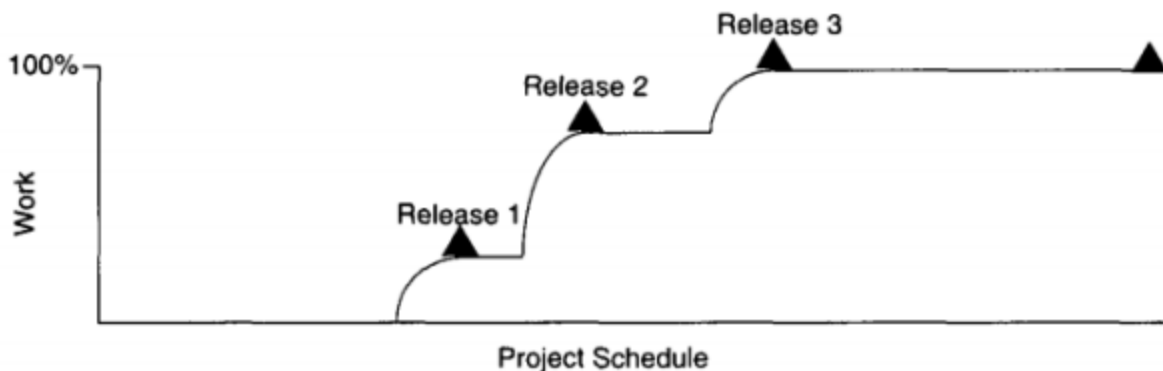
Software architecture team: - Use cases demonstrated.

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

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

Software management team: - milestones completed.

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



Expected progress for a typical project with three major releases

1.2.1.2 Budgeted cost and expenditures

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

Expenditure Plan - It is the planned spending profile for a project over its planned schedule.

Actual progress - It is the technical accomplishment relative to the planned progress underlying the spending profile.

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

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

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

Schedule variance: It is the difference between the planned cost and the earned value.

Of all parameters in an earned value system, actual progress is the most subjective

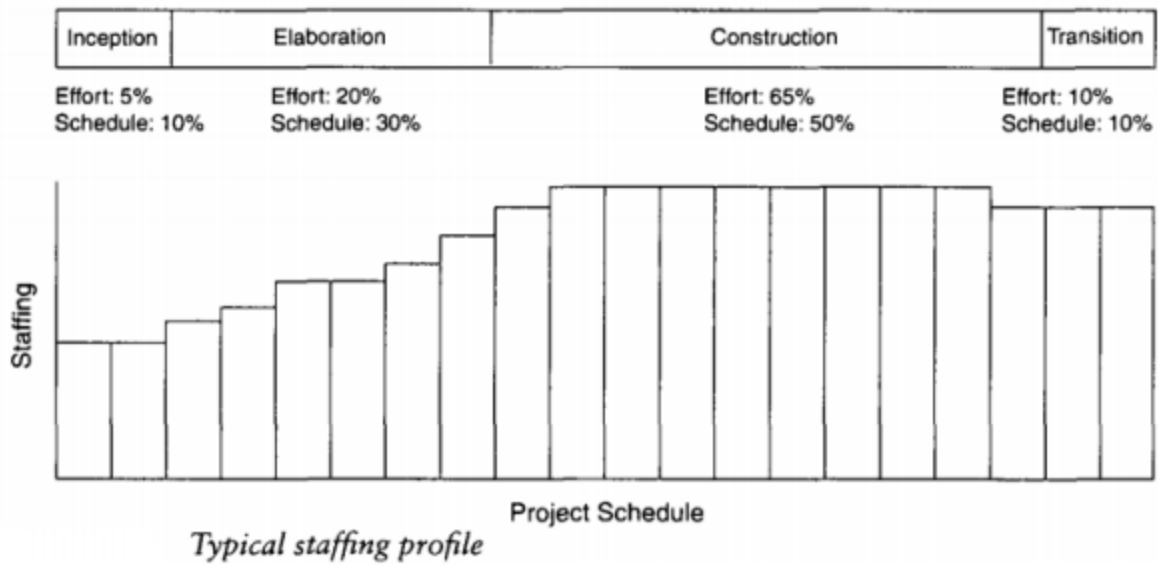
Assessment: Because most managers know exactly how much cost they have incurred and how much schedule they have used, the variability in making accurate assessments is centered in the actual progress assessment. The default perspectives of this metric are cost per month, full-time staff per month and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

1.2.1.3 Staffing and team dynamics

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

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

work and progress

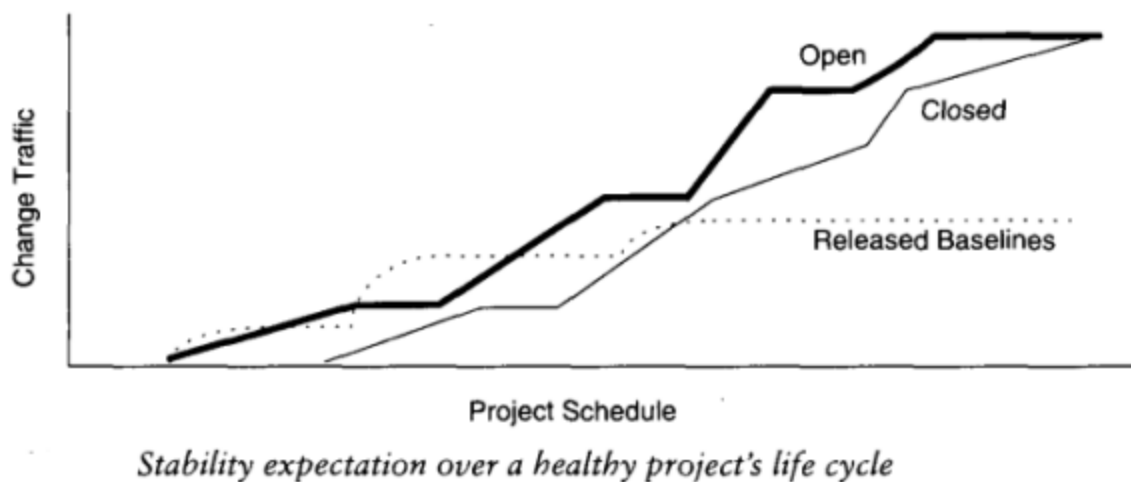


1.2.2 QUALITY INDICATORS:

1.2.2.1 Change traffic and stability:

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

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



1.2.2.2 Breakage and modularity

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

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

1.2.2.3 Rework and adaptability

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

1.2.2.4 MTBF and Maturity:

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

Software errors can be categorized into two types deterministic and nondeterministic. Deterministic errors are also known as Bohr-bugs and nondeterministic errors are also called as Heisen-bugs. Bohr-bugs are a class of errors caused when the software is stimulated in a certain way such as coding errors. Heisen-bugs are software faults that are coincidental with a certain probabilistic occurrence of a given situation, such as design errors. This metric can be collected by failure counts, test hours until failure, by release, by components and by subsystems.

These four quality indicators are based primarily on the measurement of software change across evolving baselines of engineering data.

1.3 LIFE -CYCLE EXPECTATIONS:

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

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

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

The actual values of these metrics can vary widely across projects, organizations, and domains. The relative trends across the project phases, however, should follow the general pattern shown in Table 13-3. A mature development organization should be able to describe metrics targets that are much more definitive and precise for its line of business and specific processes.

TABLE 13-3. *The default pattern of life-cycle metrics evolution*

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

1.4 METRICS AUTOMATION:

Many opportunities are available to automate the project control activities of a software project. A Software Project Control Panel (SPCP) is essential for managing against a plan.