Malla Reddy Engineering College (Autonomous)

L T P
3 - -

Course Code: 50560   Credits: 3   B.Tech.

SOFTWARE PROJECT MANAGEMENT
(Professional Elective)

Prerequisite: Software Engineering.

Objective: A basic knowledge of software project management principles and the ability to come up with a project schedule and assign resources.

Module I:Software Management

A: Conventional Software Management   [04 Periods]

The waterfall model, conventional softwareManagement performance. Overview of Project Planning — Stepwise Project Planning.

B: Improving Software Economics   [05 Periods]

Reducing Software product size, improvingsoftware processes, improving team effectiveness, improving automation, Achieving required quality, peer inspections.

Module II: SPM Phases

A: The old way and the new way   [03 Periods]

The principles of conventional software Engineering,principles of modern software management, transitioning to an iterativeprocess.

B: Life cycle phases   [03 Periods]

Engineering and production stages, Inception, E1aboration,Construction, Transition phases.

C: Artifacts of the process   [03 Periods]

The artifact sets, Management artifacts, Engineeringartifacts, programmatic artifacts.

Module III: Process

A: Work Flows of the process   [03 Periods]

Software process workflows, Iteration workflows.

B: Checkpoints of the process   [03 Periods]

Major milestones, Minor Milestones, Periodic statusassessments.

C: Iterative Process Planning   **[03** Periods]

Work breakdown structures, planning guidelines, costand schedule estimating, Iteration planning process, Pragmatic planning.

Module IV: Automation

A: Process Automation   [02 Periods]

Automation Building blocks.

B: Project Control and Process instrumentation   [05 Periods]

The seven core Metrics,Management indicators, quality indicators, life cycle expectations, pragmatic Software Metrics, Metrics automation.

C: Tailoring the Process   [02 Periods]

Process discriminants.

Module V: Responsibilities and Case Study
A: Project Organizations and Responsibilities   **[04 Periods]**
Line-of-Business Organizations, Understanding Behavior — Organizational Behavior.
B: Future Software Project Management   [03 Periods] Modern Project Profiles,
Next generationsoftware economics, modern process transitions. **C: Case Study [02 Periods]**
The command Center Processing and Display system-Replacement(CCPDS-R)

Text Books:
      1. Software Project Management, Walker Royce: Pearson Education, 2005

References:
1. Software Project Management, Bob Hughes and Mike Cotterell: Tata McGraw- Hill Edition.
2. Software Project Management, Joel Henry, Pearson Education.
3. Software Project Management in practice, Panka

Outcomes:
Upon completion of the course, the students are expected to:
1. Identify and describe how different project contexts will impact upon all aspects of a software development project
2. identify and describe the key phases of project management and the key skills associated with each
3. Determine an appropriate project management approach through an evaluation of the business context and project scope and knowledge of agile and traditional project management approaches.
4. Demonstrate through application, knowledge of the key project management skills, such as product and work break-down structure, schedule; governance including progress reporting, risk and quality management as part of a small team research and produce a concise piece of writing suitable for presentation to senior management.

Module I: Software Management
A: Conventional Software Management **[04 Periods]**
The waterfall model, conventional software Management performance. Overview of Project Planning – Stepwise Project Planning.
B: Improving Software Economics **[05 Periods]**
Reducing Software product size, improving software processes, improving team effectiveness, improving automation, Achieving required quality, peer inspections.

A. 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 manage- ment principles.

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

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

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

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

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

THE WATERFALL MODEL

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

IN THEORY

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

1. There are two essential steps common to the development of computer programs: **analysis** and **coding.**
   Waterfall Model part 1: The two basic steps to building a program.
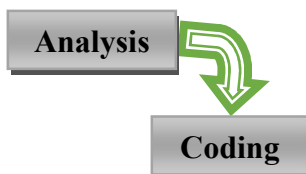
   Analysis → **Coding**

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

2. In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other "overhead" steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps. Below Figure illustrates the resulting project profile and the basic steps in developing a large-scale program.
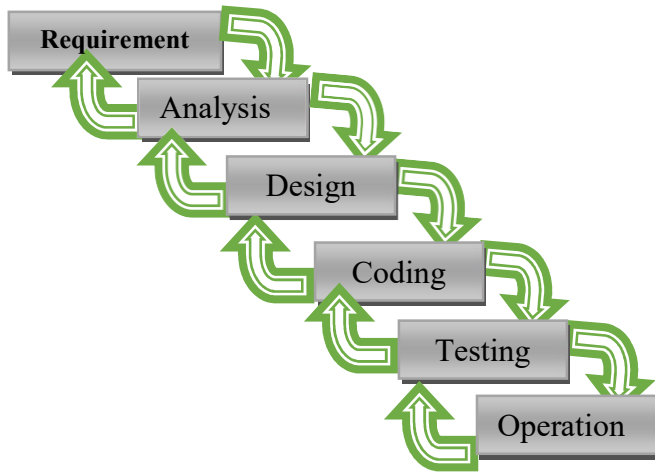
3. The basic framework described in the waterfall model is risky and invites failure. The testing phase that occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violated. Either the requirements must be modified or a substantial design change is warranted.

Five necessary improvements for waterfall model are:-

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

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

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

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

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

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

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

IN PRACTICE

Some software projects still practice the conventional software management approach.

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

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

Protracted Integration and Late Design Breakage

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

The following sequence was common:

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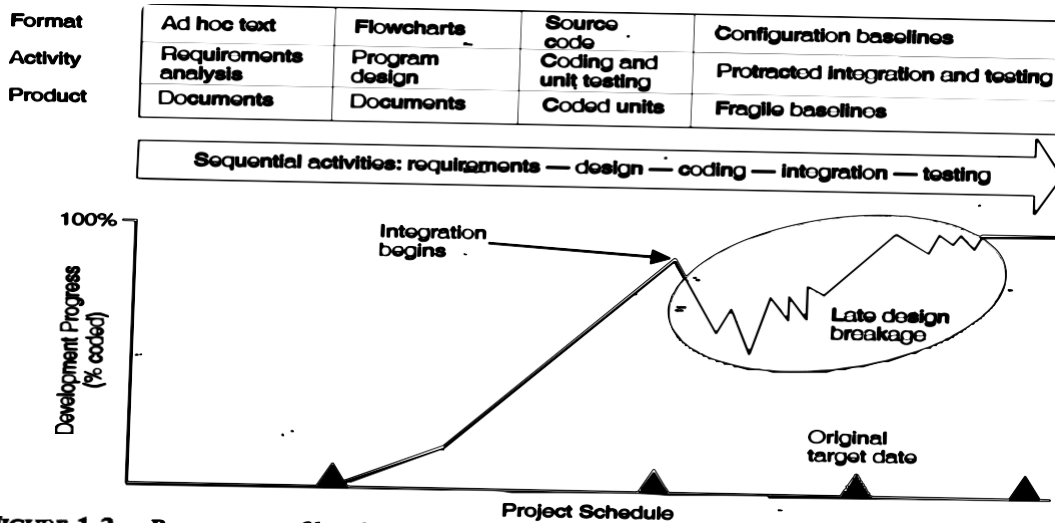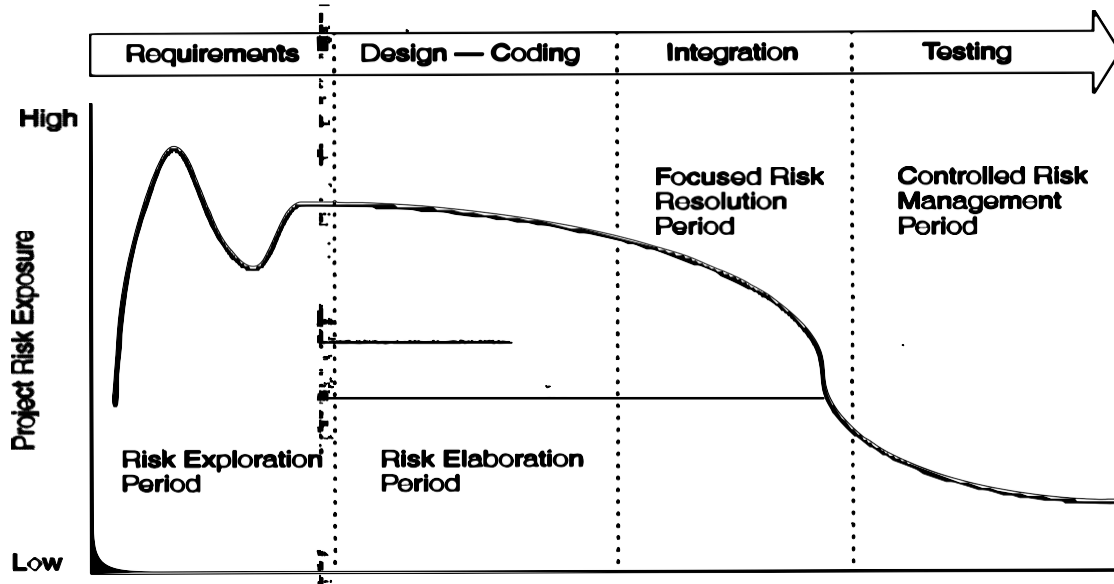- A very fragile, unmentionable product delivered late.

4

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

100%

Development Progress (% coded)

Integration begins

Late design breakage

Original target date

Project Schedule

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

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

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

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

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

| | Requirements | Design — Coding | Integration | Testing |

Project Risk Exposure (High to Low)

Risk Exploration Period · Risk Elaboration Period · Focused Risk Resolution Period · Controlled Risk Management Period

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

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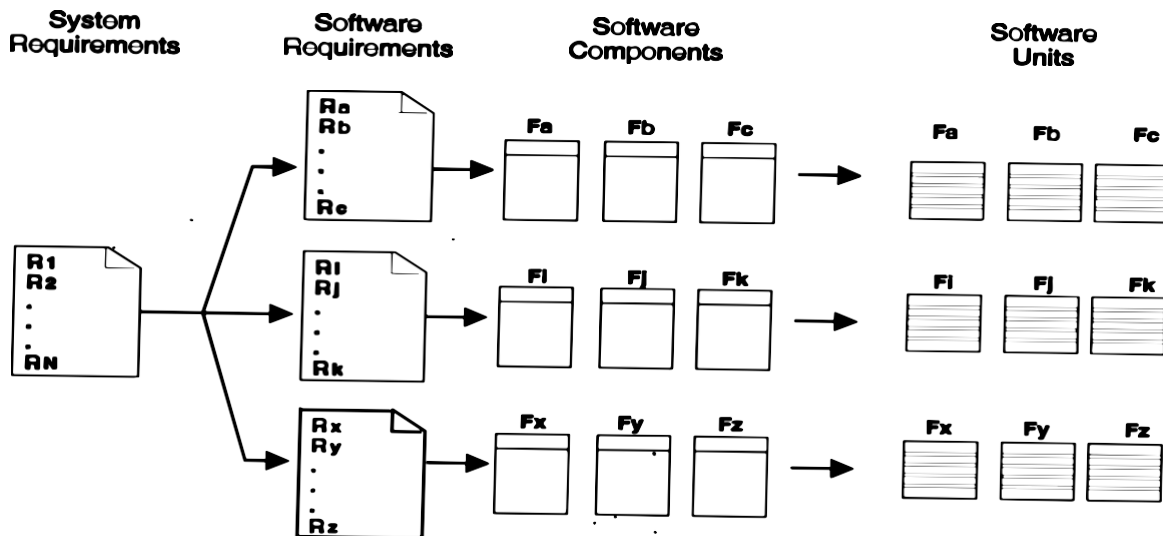

FIGURE 1-4. *Suboptimal software component organization resulting from a requirements-driven approach*

Adversarial Stakeholder Relationships:

The conventional process tended to result in adversarial stakeholder relationships, in large part because of the difficulties of requirements specification and the exchange of information solely through paper documents that captured engineering information in ad hoc formats.

6

The following sequence of events was typical for most contractual software efforts:

1. The contractor prepared a draft contract-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.

2. The customer was expected to provide comments (typically within 15 to 30 days).

3. The contractor incorporated these comments and submitted (typically within 15 to 30 days) a final version for approval.

This one-shot review process encouraged high levels of sensitivity on the part of customers and contractors.

Focus on Documents and Review Meetings:

The conventional process focused on producing various documents that attempted to describe the software product, with insufficient focus on producing tangible increments of the products themselves. Contractors were driven to produce literally tons of paper to meet milestones and demonstrate progress to stakeholders, rather than spend their energy on tasks that would reduce risk and produce quality software. Typically, presenters and the audience reviewed the simple things that they understood rather than the complex and important issues. Most design reviews therefore resulted in low engineering value and high cost in terms of the effort and schedule involved in their preparation and conduct. They presented merely a facade of progress.

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

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

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

CONVENTIONAL SOFTWARE MANAGEMENT PERFORMANCE

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

1. Finding and fixing a software problem after delivery **costs 100** times more than finding and fixing the problem in early design phases.
2. You can compress software development schedules **25%** of nominal, but no more.
3. For every **$1** you spend on development, you will spend **$2** on maintenance.
4. Software development and maintenance costs are primarily a function of the number of source lines of code.
5. Variations among people account for the **biggest** differences in software productivity.
6. The overall ratio of software to hardware costs is still growing. In 1955 it was **15:85**; in 1985, **85:15**.
7. Only about **15%** of software development effort is devoted to programming.
8. Software systems and products typically cost 3 times as much per SLOC as individual software programs. Software-system products (i.e., system of systems) cost 9 times as much.
9. Walkthroughs catch **60%** of the errors

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

Overview of Project Planning

After the project has been defined and the project team has been appointed, you are ready to enter the second phase in the project management life cycle: the detailed project planning phase.

Project planning is at the heart of the project life cycle, and tells everyone involved where you're going and how you're going to get there. The planning phase is when the project plans are documented, the project deliverables and requirements are defined, and the project schedule is created. It involves creating a set of plans to help guide your team through the implementation and closure phases of the project. The plans created during this phase will help you manage time, cost, quality, changes, risk, and related issues. They will also help you control staff and external suppliers to ensure that you deliver the project on time, within budget, and within schedule.

The project planning phase is often the most challenging phase for a project manager, as you need to make an educated guess about the staff, resources, and equipment needed to complete your project. You may also need to plan your communications and procurement activities, as well as contract any third-party suppliers.

The purpose of the project planning phase is to:
- Establish business requirements
- Establish cost, schedule, list of deliverables, and delivery dates
- Establish resources plans
- Obtain management approval and proceed to the next phase

The basic processes of project planning are:
- Scope planning – specifying the in-scope requirements for the project to facilitate creating the work breakdown structure
- Preparation of the work breakdown structure – spelling out the breakdown of the project into tasks and sub-tasks
- Project schedule development – listing the entire schedule of the activities and detailing their sequence of implementation
- Resource planning – indicating who will do what work, at which time, and if any special skills are needed to accomplish the project tasks
- Budget planning – specifying the budgeted cost to be incurred at the completion of the project
- Procurement planning – focusing on vendors outside your company and subcontracting
- Risk management – planning for possible risks and considering optional contingency plans and mitigation strategies
- Quality planning – assessing quality criteria to be used for the project
- Communication planning – designing the communication strategy with all project stakeholders

The planning phase refines the project's objectives, which were gathered during the initiation phase. It includes planning the steps necessary to meet those objectives by further identifying the specific activities and resources required to complete the project. Now that these objectives have been recognized, they must be clearly articulated, detailing an in-depth scrutiny of each recognized objective. With such scrutiny, our understanding of the objective may change. Often the very act of trying to describe something precisely gives us a better understanding of what we are looking at. This articulation serves as the basis for the development of requirements. What this means is that after an objective has been clearly articulated, we can describe it in concrete (measurable) terms and identify what we have to do to achieve it. Obviously, if we do a poor job of articulating the objective, our requirements will be misdirected and the resulting project will not represent the true need.

Users will often begin describing their objectives in qualitative language. The project manager must work with the user to provide quantifiable definitions to those qualitative terms. These quantifiable criteria include schedule, cost, and quality measures. In the case of project objectives, these elements are used as measurements to determine project satisfaction and successful completion. Subjective evaluations are replaced by actual numeric attributes.

Example 1
A web user may ask for a fast system. The quantitative requirement should be all screens must load in under three seconds. Describing the time limit during which the screen must load is specific and tangible. For that reason, you'll know that the requirement has been successfully completed when the objective has been met.

Example 2
Let's say that your company is going to produce a holiday batch of eggnog. Your objective statement might be stated this way: Christmas Cheer, Inc. will produce two million cases of holiday eggnog, to be shipped to our distributors by October 30, at a total cost of $1.5 million or less. The objective criteria in this statement are clearly stated and successful fulfillment can easily be measured. Stakeholders will know that the objectives are met when the two million cases are produced and shipped by the due date within the budget stated.

When articulating the project objectives you should follow the SMART rule:

- Specific – get into the details. Objectives should be specific and written in clear, concise, and understandable terms.
- Measurable – use quantitative language. You need to know when you have successfully completed the task.
- Acceptable – agreed with the stakeholders.
- Realistic – in terms of achievement. Objectives that are impossible to accomplish are not realistic and not attainable. Objectives must be centred in reality.
- Time based – deadlines not durations. Objectives should have a time frame with an end date assigned to them.

If you follow these principles, you'll be certain that your objectives meet the quantifiable criteria needed to measure success.

Stepwise Project Planning
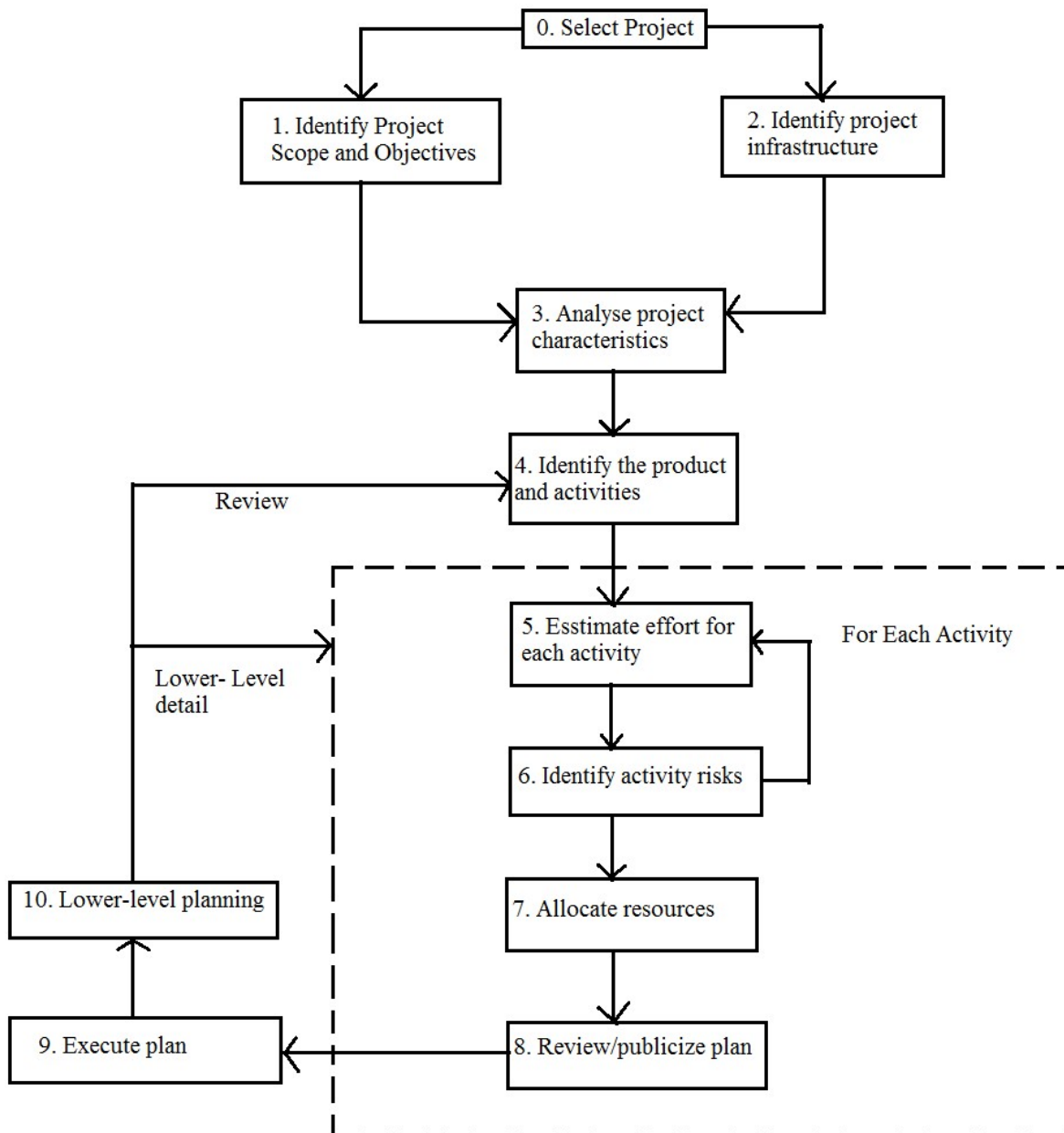
Planning is the most difficult process in project management. The framework described is called the Stepwise method to help to distinguish it from other methods.

Step 0: Select Project

Step 1: Identify project scope and objectives
Step 1.1 : Identify objectives and practical measures of the effectiveness in meeting those objectives
Step 1.2 : Establish a project authority
Step 1.3 : Stakeholder analysis - identify all stakeholders in the project and their interests.
Step 1.4 : Modify objectives in the light of stakeholder analysis.
Step 1.5 : Establish methods of communication with all parties.

Step 2 : Identify project infrastructure
Step 2.2 : Identify installation standard and procedures

Step 2.3 : Identify project team organization

Step 3 : Analyse project characteristics
Step 3.1 : Distinguish the project as either objectives- or product-driven.
Step 3.2 : Analyse other project characteristics
Step 3.3 : Identify high-level project risks
Step 3.4 : Take into account use requirements concerning implementation
Step 3.5 : Select development methodology and life-cycle approach
Step 3.6 : Review overall resource estimates

Step 4 : Identify project products and activities
Step 4.1 : Identify and describe project products
Step 4.2 : Document generic product flows
Step 4.3 : Recognize product instances
Step 4.4 : Produce ideal activity network
Step 4.5 : Modify the ideal to take into account need for stages and checkpoints

Step 5 : Estimate effort for each activity
Step 5.1 : Carry out bottom-up estimates
      - distinguish carefully between effort and elapsed time
Step 5.2 : Revise plan to create ontrollable activities
         - breakup very long activities into a series of smaller ones
         - bundle up very short activities

Step 6 : Identify activity risks
Step 6.1 : Identify and quantify activity based risks
         - damage if risk occurs
         - likelihood if risk occuring
Step 6.2 : Plan risk reduction and contingency measures
         - risk reduction : activity to stop risk occuring
         - contingency : action if risk does occurs
Step 6.3 : Adjust overall plans and estimates to take account of risks

Step 7 : Allocate resources
Step 7.1 : Identify and allocate resources
Step 7.2 : Revise plans and estimates to take into account resource constraints

Step 8 : Review/ Publicize plans
Step 8.1 : Review quality aspects of the project plan
Step 8.2 : Documentr plans and obtain agreement

Step 9 and 10 : Execute plan. Lower levels of planning

B: Improving Software Economics
    Five basic parameters of the software cost model are

       1. Reducing the *size* or complexity of what needs to be developed.

       2. Improving the development *process.*

       3. Using more-skilled *personnel* and better teams (not necessarily the same thing).

4. Using better **environments** (tools to automate the process).

5. Trading off or backing off on **quality** thresholds.

These parameters are given in priority order for most software domains. Table 3-1 lists some of the technology developments, process improvement efforts, and management approaches targeted at improving the economics of software development and integration.

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

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

REDUCING SOFTWARE PRODUCT SIZE

The most significant way to improve affordability and return on investment (ROI) is usually to produce a product that achieves the design goals with the minimum amount of human-generated source material. ***Component-based development*** is introduced as the general term for reducing the "**source**" language size to achieve a software solution.

Reuse, object-oriented technology, automatic code production, and higher order programming languages are all focused on achieving a given system with fewer lines of human-specified source directives (statements).

size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic), automatic code generators (CASE tools, visual modeling tools, GUI builders), reuse of **commercial components** (operating systems, windowing environments, database management systems, middleware, networks), and object-oriented technologies (Unified Modeling Language, visual modeling tools, architecture frameworks).

The reduction is defined in terms of human-generated source material. In general, when size-reducing technologies are used, they reduce the number of human-generated source lines.

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

Languages expressiveness of some of today's popular languages

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

Table 3-2

OBJECT-ORIENTED METHODS AND VISUAL MODELING

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

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

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

1. An object-oriented model of the problem and its solution encourages a common vocabulary between the end users of a system and its developers, thus creating a shared understanding of the problem being solved.

2. The use of continuous integration creates opportunities to recognize risk early and make incremental corrections without destabilizing the entire development effort.

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

---

[1] Function point metrics provide a standardized method for measuring the various functions of a software application.
The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries.

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

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

REUSE

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

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

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

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

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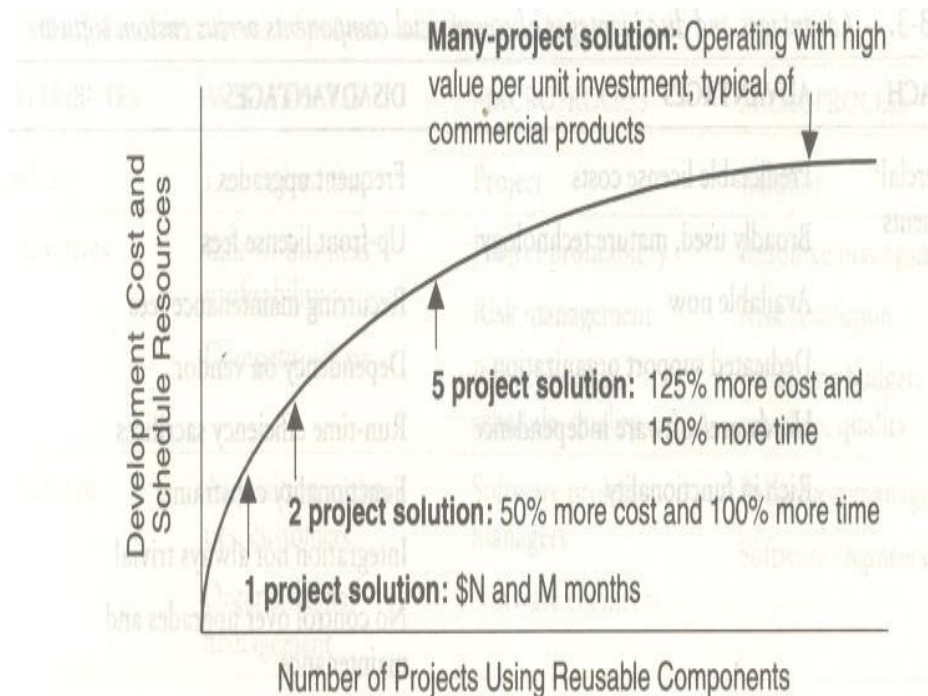

Many-project solution: Operating with high value per unit investment, typical of commercial products

5 project solution: 125% more cost and 150% more time

2 project solution: 50% more cost and 100% more time

1 project solution: $N and M months

Development Cost and Schedule Resources

Number of Projects Using Reusable Components

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

COMMERCIAL COMPONENTS

A common approach being pursued today in many domains is to maximize integration of commercial components and off-the-shelf products. While the use of commercial components is certainly desirable as a means of reducing custom development, it has not proven to be straightforward in practice. Table 3-3 identifies some of the advantages and disadvantages of using commercial components.

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

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

IMPROVING SOFTWARE PROCESSES

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

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

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

16

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

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

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

IMPROVING TEAM EFFECTIVENESS

Teamwork is much more important than the sum of the individuals. With software teams, a project manager needs to configure a balance of solid talent with highly skilled people in the leverage positions. Some maxims of team management include the following:

- A well-managed project can succeed with a nominal engineering team.
- A mismanaged project will almost never succeed, even with an expert team of engineers.
- A well-architected system can be built by a nominal team of software builders.

17

- A poorly architected system will flounder even with an expert team of builders.

1. The principle of top talent: Use better and fewer people
2. The principle of job matching: Fit the tasks to the skills and motivation of the people available.
3. The principle of career progression: An organization does best in the long run by helping its people to **self-actualize**.
4. The principle of team balance: Select people who will complement and harmonize with one another
5. The principle of phase-out: Keeping a misfit on the team doesn't benefit anyone

Software project managers need many leadership qualities in order to enhance team effectiveness. The following are some crucial attributes of successful software project managers that deserve much more attention:

1. **Hiring skills**. Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.
2. **Customer-interface skill**. Avoiding adversarial relationships among stakeholders is a prerequisite for success.

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

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

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

IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

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

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

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

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

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

companies in a particular tool.

- Requirements analysis and evolution activities consume 40% of life-cycle costs.

- Software design activities have an impact on more than 50% of the resources.

- Coding and unit testing activities consume about 50% of software development effort and schedule.

- Test activities can consume as much as 50% of a project's resources.
- Configuration control and change management are critical activities that can consume as much as 25% of resources on a large-scale project.
- Documentation activities can consume more than 30% of project engineering resources.
- Project management, business administration, and progress assessment can consume as much as 30% of project budgets.

ACHIEVING REQUIRED QUALITY

Software best practices are derived from the development process and technologies. Table 3-5 summarizes some dimensions of quality improvement.

Key practices that improve overall software quality include the following:

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

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

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

Conventional development processes stressed early sizing and timing estimates of computer program resource utilization. However, the typical chronology of events in performance assessment was as follows

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

PEER INSPECTIONS: A PRAGMATICVIEW

Peer inspections are frequently over hyped as the key aspect of a quality system. In my experience, peer reviews are valuable as secondary mechanisms, but they are rarely significant contributors to quality compared with the following primary quality mechanisms and indicators, which should be emphasized in the management process:

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

Inspections are also a good vehicle for holding authors accountable for quality products. All authors of software and documentation should have their products scrutinized as a natural by-product of the process. Therefore, the coverage of inspections should be across all authors rather than across all components.

Module II: SPM Phases
A: The old way and the new way [03 Periods]
The principles of conventional software Engineering, principles of modern software management, transitioning to an iterative process.
B: Life cycle phases [03 Periods]
Engineering and production stages, Inception, Elaboration, Construction, Transition phases.
C: Artifacts of the process [03 Periods]
The artifact sets, Management artifacts, Engineering artifacts, programmatic artifacts.


The old way and the new way

THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

1. **Make quality** Quality must be quantified and mechanisms put into place to motivate its achievement

2. **High-quality software is possible**. Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people

3. **Give products to customers early**. No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it

4. **Determine the problem before writing the requirements**. When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution

5. **Evaluate design alternatives**. After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use" architecture" simply because it was used in the requirements specification.

6. **Use an appropriate process model**. Each project must select a process that makes ·the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.

7. **Use different languages for different phases**. Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation through-out the life cycle.

8. **Minimize intellectual distance**. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure

9. **Put techniques before tools**. An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer

10. **Get it right before you make it faster**. It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during initial coding

11. **Inspect code**. Inspecting the detailed design and code is a much better way to find errors than testing

12. **Good management is more important than good technology**. Good management motivates people to do their best, but there are no universal "right" styles of management.

13. **People are the key to success.** Highly skilled people with appropriate experience, talent, and training are key.

14. **Follow with care**. Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.

15. *Take responsibility*. When a bridge collapses we ask, "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.

16. **Understand the customer's priorities**. It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.

17. **The more they see, the more they need**. The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.

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

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

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

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

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

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

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

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

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

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

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

29. **People and time are not interchangeable**. Measuring a project solely by person-months makes little sense

30. **Expect excellence**. Your employees will do much better if you have high expectations for them.

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

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

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

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

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

FIGURE 4-1. *The top five principles of a modern process*
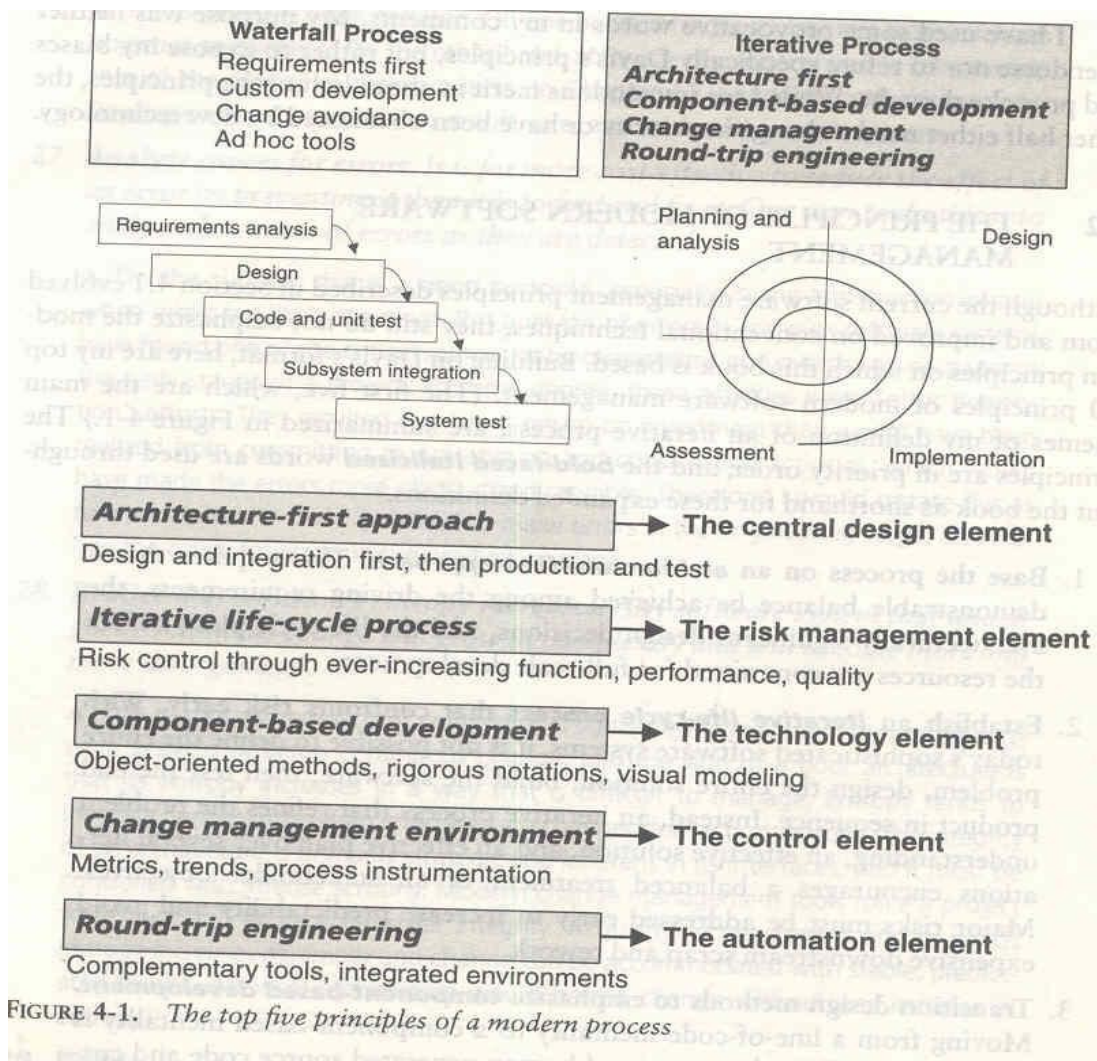
5. Enhance change freedom through tools that support round-trip engineering. Round-trip engineering is the environment support necessary to automate and synchronize engineering information in different formats(such as requirements specifications, design models, source code, executable code, test cases).

6. Capture design artifacts in rigorous, model-based notation. A model based approach (such as UML) supports the evolution of semantically rich graphical and textual design notations.

7. Instrument the process for objective quality control and progress assessment. Life-cycle assessment of the progress and the quality of all intermediate products must be integrated into the process.

8. **Use a demonstration-based approach to assess intermediate artifacts.**

9. **Plan intermediate releases in groups of usage scenarios with evolving levels of detail.** It is essential that the software management process drive toward early and continuous demonstrations within the operational context of the system, namely its use cases.

10. **Establish a configurable process that is economically scalable.** No single process is suitable for all software developments.

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

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

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

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

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

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

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

- **Process flexibility**. Development of modern software is characterized by such a broad solution space and so many interrelated concerns that there is a paramount need for continuous incorporation of changes. These changes may be inherent in the problem understanding, the solution space, or the plans. Project artifacts must be supported by efficient *change management* commensurate with project needs. A *configurable process* that allows a common framework to be adapted across a range of projects is necessary to achieve a software return on investment.

- **Architecture risk resolution**. *Architecture-first* development is a crucial theme underlying a successful iterative development process. A project team develops and stabilizes architecture before developing all the components that make up the entire suite of applications components. An *architecture-first* and *component-based development approach* forces the infrastructure, common mechanisms, and control mechanisms to be elaborated early in the life cycle and drives all component make/buy decisions into the architecture process.

- **Team cohesion**. Successful teams are cohesive, and cohesive teams are successful. Successful teams and cohesive teams share common objectives and priorities. Advances in technology (such as programming languages, UML, and visual modeling) have enabled more rigorous and understandable notations for communicating software engineering information, particularly in the requirements and design artifacts that previously were ad hoc and based completely on paper exchange. These *model-based* formats have also enabled the *round-trip engineering* support needed to establish change freedom sufficient for evolving design representations.

- **Software process maturity**. The Software Engineering Institute's Capability Maturity Model (CMM) is a well-accepted benchmark for software process assessment. One of key themes is that truly mature processes are enabled through an integrated environment that provides the appropriate level of automation to instrument the process for *objective quality control.*

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*

B: Life cycle phases

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

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

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

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

- Evolution of the plans, requirements, and architecture, together with well defined synchronization points
- Risk management and objective measures of progress and quality
- Evolution of system capabilities through demonstrations of increasing functionality

ENGINEERING AND PRODUCTION STAGES

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

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

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

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

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

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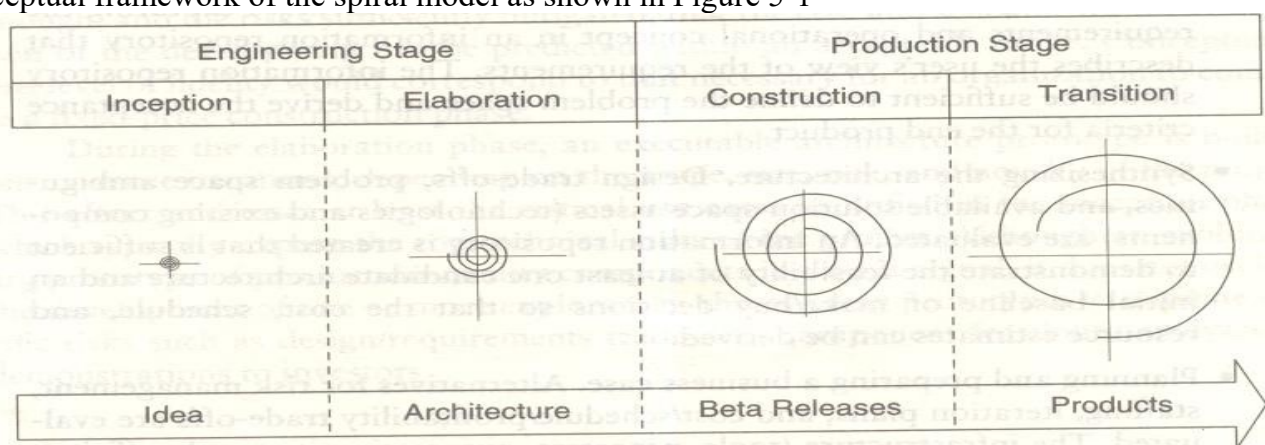

	Engineering Stage		Production Stage	
	Inception	Elaboration	Construction	Transition
	Idea	Architecture	Beta Releases	Products

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

INCEPTION PHASE

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

PRIMARY OBJECTIVES

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

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

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

ELABORATION PHASE

At the end of this phase, the "engineering" is considered complete. The elaboration phase activities must ensure that the architecture, requirements, and plans are stable enough, and the risks sufficiently mitigated, that the cost and schedule for the completion of the development can be predicted within an acceptable range. During the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size, & risk.

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

ESSENTIAL ACTIVITIES
- Elaborating the vision.
- Elaborating the process and infrastructure.
- Elaborating the architecture and selecting components.

PRIMARY EVALUATION CRITERIA
- Is the vision stable?
- Is the architecture stable?
- Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?
- Is the construction phase plan of sufficient fidelity, and is it backed up with a credible basis of estimate?
- Do all stakeholders agree that the current vision can be met if the current plan is executed to develop the

complete system in the context of the current architecture?

- Are actual resource expenditures versus planned expenditures acceptable?

CONSTRUCTION PHASE

During the construction phase, all remaining components and application features are integrated into the application, and all features are thoroughly tested. Newly developed software is integrated where required. The construction phase represents a production process, in which emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality.

PRIMARY OBJECTIVES

- Minimizing development costs by optimizing resources and avoiding unnecessary scrap and rework
- Achieving adequate quality as rapidly as practical
- Achieving useful versions (alpha, beta, and other test releases) as rapidly as practical

ESSENTIAL ACTIVITIES

- Resource management, control, and process optimization
- Complete component development and testing against evaluation criteria
- Assessment of product releases against acceptance criteria of the vision

PRIMARY EVALUATION CRITERIA

- Is this product baseline mature enough to be deployed in the user community? (Existing defects are not obstacles to achieving the purpose of the next release.)
- Is this product baseline stable enough to be deployed in the user community? (Pending changes are not obstacles to achieving the purpose of the next release.)
- Are the stakeholders ready for transition to the user community?
- Are actual resource expenditures versus planned expenditures acceptable?

TRANSITION PHASE

The transition phase is entered when a baseline is mature enough to be deployed in the end-user domain. This typically requires that a usable subset of the system has been achieved with acceptable quality levels and user documentation so that transition to the user will provide positive results. This phase could include any of the following activities:

1. Beta testing to validate the new system against user expectations
2. Beta testing and parallel operation relative to a legacy system it is replacing
3. Conversion of operational databases
4. Training of users and maintainers

The transition phase concludes when the deployment baseline has achieved the complete vision.

PRIMARY OBJECTIVES

- Achieving user self-supportability
- Achieving stakeholder concurrence that deployment baselines are complete and consistent with the evaluation criteria of the vision
- Achieving final product baselines as rapidly and cost-effectively as practical

ESSENTIAL ACTIVITIES

- Synchronization and integration of concurrent construction increments into consistent deployment baselines
- Deployment-specific engineering (cutover, commercial packaging and production, sales rollout kit development, field personnel training)
- Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set

EVALUATION CRITERIA

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

C: Artifacts of the process

THE ARTIFACT SETS

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

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

Requirements Set	Design Set	Implementation Set	Deployment Set
1. Vision document 2. Requirements model(s)	1. Design model(s) 2. Test model 3. Software architecture description	1. Source code baselines 2. Associated compile-time files 3. Component executables	1. Integrated product executable baselines 2. Associated run-time files 3. User manual

Management Set	
Planning Artifacts	**Operational Artifacts**
1. Work breakdown structure 2. Business case 3. Release specifications 4. Software development plan	5. Release descriptions 6. Status assessments 7. Software change order database 8. Deployment documents 9. Environment

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

THE MANAGEMENT SET

The management set captures the artifacts associated with process planning and execution. These artifacts use ad hoc notations, including text, graphics, or whatever representation is required to capture the "contracts" among project personnel (project management, architects, developers, testers, marketers, administrators), among stakeholders (funding authority, user, software project manager, organization manager, regulatory agency), and between project personnel and stakeholders. Specific

artifacts included in this set are the work breakdown structure (activity breakdown and financial tracking mechanism), the business case (cost, schedule, profit expectations), the release specifications (scope, plan, objectives for release baselines), the software development plan (project process instance), the release descriptions (results of release baselines), the status assessments (periodic snapshots of project progress), the software change orders (descriptions of discrete baseline changes), the deployment documents (cutover plan, training course, sales rollout kit), and the environment (hardware and software tools, process automation, & documentation).

Management set artifacts are evaluated, assessed, and measured through a combination of the following:
- Relevant stakeholder review
- Analysis of changes between the current version of the artifact and previous versions
- Major milestone demonstrations of the balance among all artifacts and, in particular, the accuracy of the business case and vision artifacts

THE ENGINEERING SETS

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

Requirements Set

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

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

Design Set

UML notation is used to engineer the design models for the solution. The design set contains varying levels of abstraction that represent the components of the solution space (their identities, attributes, static relationships, dynamic interactions). The design set is evaluated, assessed, and measured through a combination of the following:
- Analysis of the internal consistency and quality of the design model
- Analysis of consistency with the requirements models
- Translation into implementation and deployment sets and notations (for example, traceability, source code generation, compilation, linking) to evaluate the consistency and completeness and the semantic balance between information in the sets
- Analysis of changes between the current version of the design model and previous versions (scrap, rework, and defect elimination trends)
- Subjective review of other dimensions of quality

Implementation set

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

Implementation sets are human-readable formats that are evaluated, assessed, and measured through a combination of the following:
- Analysis of consistency with the design models
- Translation into deployment set notations (for example, compilation and linking) to evaluate the consistency and completeness among artifact sets
- Assessment of component source or executable files against relevant evaluation criteria through inspection, analysis, demonstration, or testing
- Execution of stand-alone component test cases that automatically compare expected results with actual results
- Analysis of changes between the current version of the implementation set and previous versions

(scrap, rework, and defect elimination trends)
- Subjective review of other dimensions of quality

Deployment Set

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

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

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

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

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

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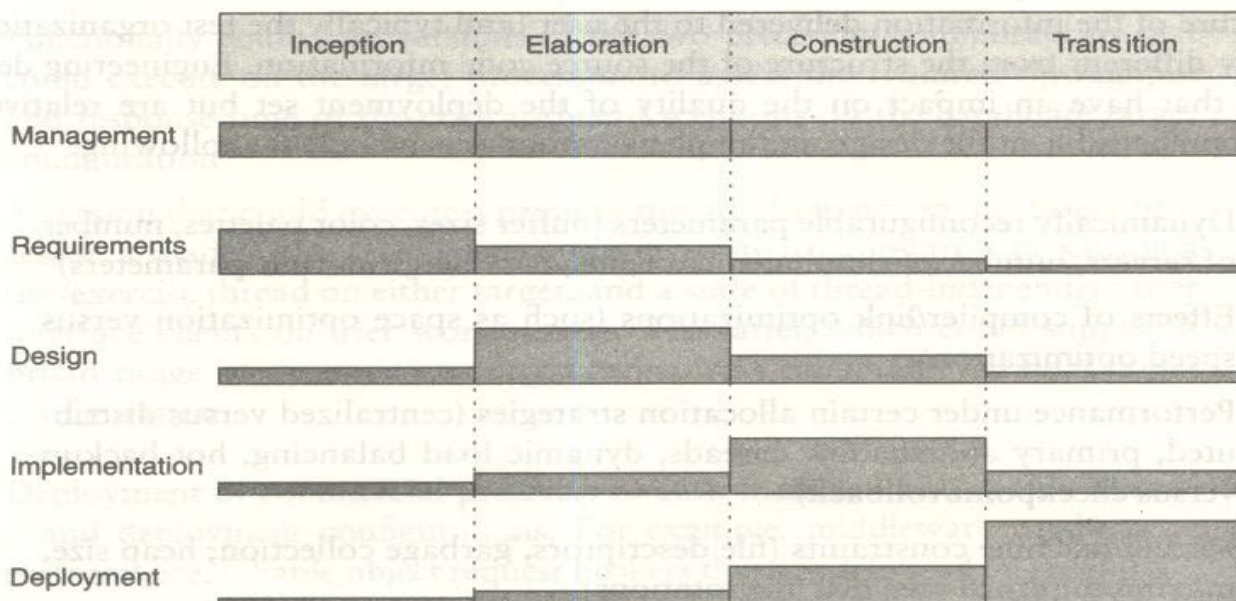

FIGURE 6-2. *Life-cycle focus on artifact sets*

Implementation Set versus Deployment Set

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

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

ARTIFACT EVOLUTION OVER THE LIFE CYCLE

Each state of development represents a certain amount of precision in the final system description. Early in the life cycle, precision is low and the representation is generally high. Eventually, the precision of representation is high and everything is specified in full detail. Each phase of development focuses on a particular artifact set. At the end of each phase, the overall system state will have progressed on all sets, as illustrated in Figure 6-3.
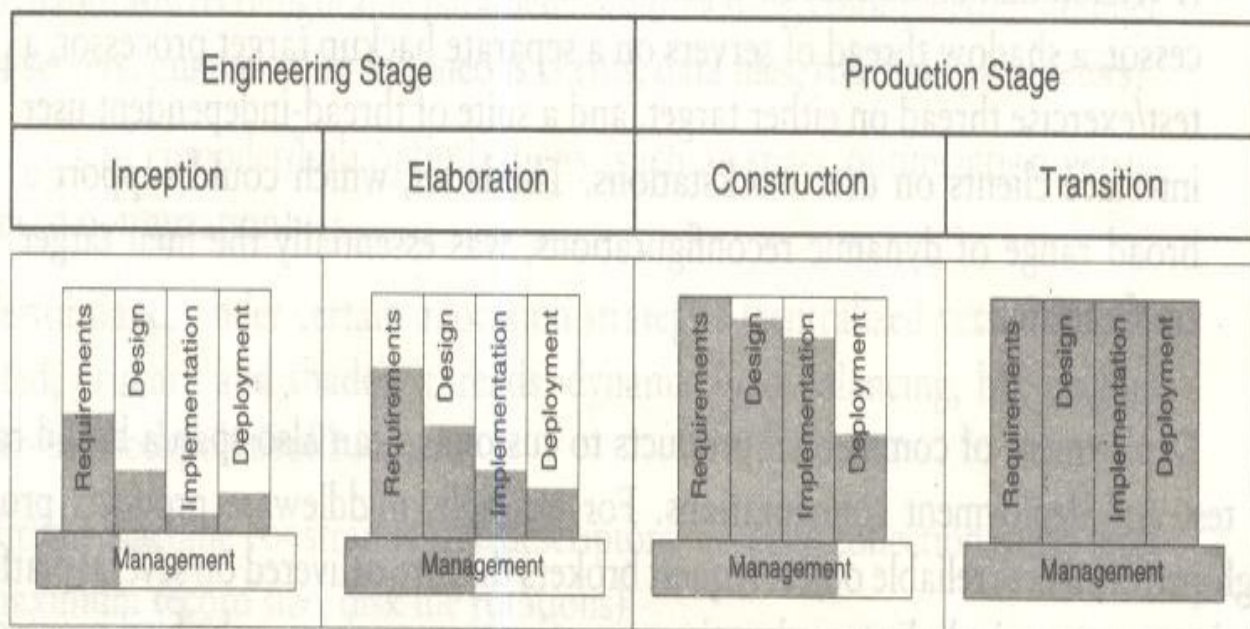
FIGURE 6-3. *Life-cycle evolution of the artifact sets*

The **inception** phase focuses mainly on critical requirements usually with a secondary focus on an initial deployment view. During the **elaboration phase**, there is much greater depth in requirements, much more breadth in the design set, and further work on implementation and deployment issues. The main focus of the **construction** phase is design and implementation. The main focus of the **transition** phase is on achieving consistency and completeness of the deployment set in the context of the other sets.

- The test artifacts must be developed concurrently with the product from inception through deployment. Thus, testing is a full-life-cycle activity, not a late life-cycle activity.
- The test artifacts are communicated, engineered, and developed within the same artifact sets as the developed product.
- The test artifacts are implemented in programmable and repeatable formats (as software programs).
- The test artifacts are documented in the same way that the product is documented.
- Developers of the test artifacts use the same tools, techniques, and training as the software engineers developing the product.

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

- Management set. The release specifications and release descriptions capture the objectives, evaluation criteria, and results of an intermediate milestone. These artifacts are the test plans and test results negotiated among internal project teams. The software change orders capture test results (defects, testability changes, requirements ambiguities, enhancements) and the closure criteria associated with making a discrete change to a baseline.
- Requirements set. The system-level use cases capture the operational concept for the system and the acceptance test case descriptions, including the expected behavior of the system and its quality attributes. The entire requirement set is a test artifact because it is the basis of all assessment activities across the life cycle.
- Design set. A test model for nondeliverable components needed to test the product baselines is captured in the design set. These components include such design set artifacts as a seismic event simulation for creating realistic sensor data; a "virtual operator" that can support unattended, after-hours test cases; specific instrumentation suites for early demonstration of resource usage; transaction rates or response times; and use case test drivers and component stand-alone test drivers.
- Implementation set. Self-documenting source code representations for test components and test drivers provide the equivalent of test procedures and test scripts. These source files may also include human-readable data files representing certain statically defined data sets that are explicit test source files. Output files from test drivers provide the equivalent of test reports.
- Deployment set. Executable versions of test components, test drivers, and data files are provided.

MANAGEMENT ARTIFACTS

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

Business Case

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

cycle progresses. Figure 6-4

provides a default outline for a business case.

Software Development Plan

The software development plan (SDP) elaborates the process framework into a fully detailed plan. Two indications of a useful SDP are periodic updating (it is not stagnant shelfware) and understanding and acceptance by managers and practitioners alike. Figure 6-5 provides a default outline for a software development plan.
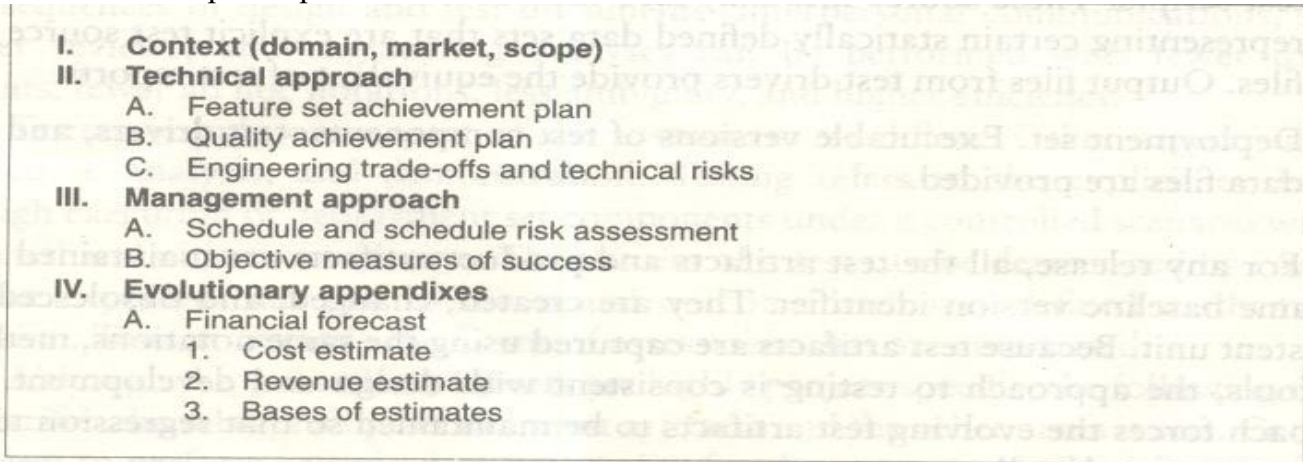
```
I.    Context (domain, market, scope)
II.   Technical approach
      A.  Feature set achievement plan
      B.  Quality achievement plan
      C.  Engineering trade-offs and technical risks
III.  Management approach
      A.  Schedule and schedule risk assessment
      B.  Objective measures of success
IV.   Evolutionary appendixes
      A.  Financial forecast
          1.  Cost estimate
          2.  Revenue estimate
          3.  Bases of estimates
```

FIGURE 6-4.   *Typical business case outline*
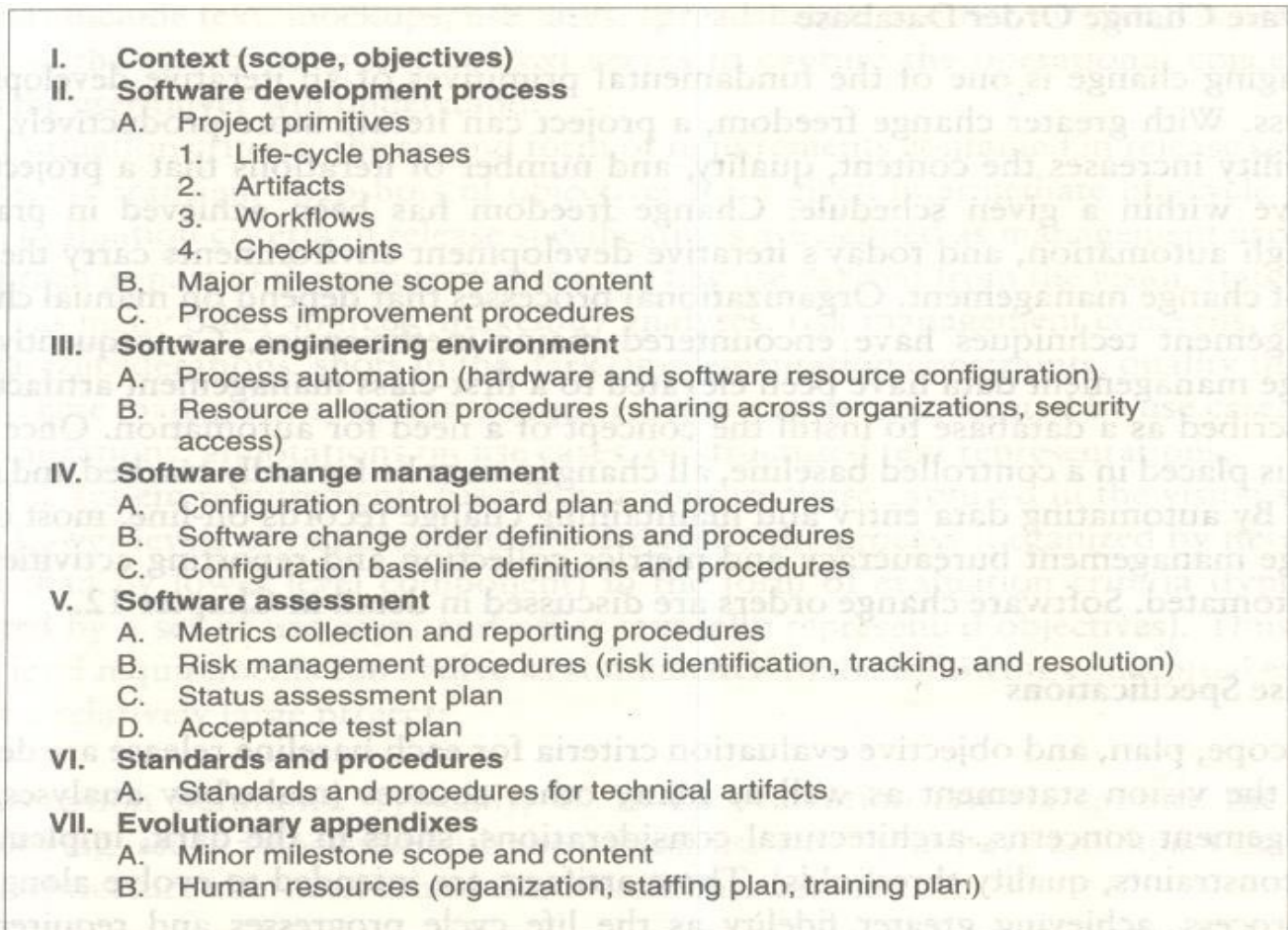
```
I.     Context (scope, objectives)
II.    Software development process
       A.  Project primitives
           1.  Life-cycle phases
           2.  Artifacts
           3.  Workflows
           4.  Checkpoints
       B.  Major milestone scope and content
       C.  Process improvement procedures
III.   Software engineering environment
       A.  Process automation (hardware and software resource configuration)
       B.  Resource allocation procedures (sharing across organizations, security
           access)
IV.    Software change management
       A.  Configuration control board plan and procedures
       B.  Software change order definitions and procedures
       C.  Configuration baseline definitions and procedures
V.     Software assessment
       A.  Metrics collection and reporting procedures
       B.  Risk management procedures (risk identification, tracking, and resolution)
       C.  Status assessment plan
       D.  Acceptance test plan
VI.    Standards and procedures
       A.  Standards and procedures for technical artifacts
VII.   Evolutionary appendixes
       A.  Minor milestone scope and content
       B.  Human resources (organization, staffing plan, training plan)
```

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

Work Breakdown Structure

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

Software Change Order Database

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

Release Specifications

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

```
I.      Iteration content
II.     Measurable objectives
        A.  Evaluation criteria
        B.  Followthrough approach
III.    Demonstration plan
        A.  Schedule of activities
        B.  Team responsibilities
IV.     Operational scenarios (use cases demonstrated)
        A.  Demonstration procedures
        B.  Traceability to vision and business case
```
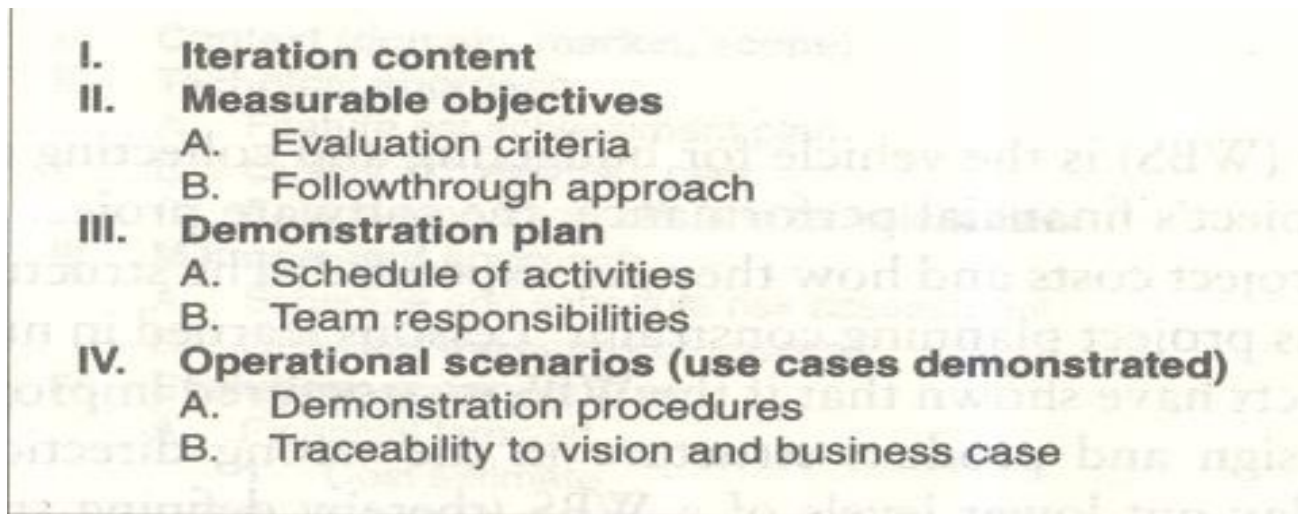
FIGURE 6-6.   *Typical release specification outline*

Release Descriptions

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

Status Assessments

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

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

FIGURE 6-7.   *Typical release description outline*

Environment

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

Deployment

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

Management Artifact Sequences

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

Legend:
- △ Informal version
- ▲ Controlled baseline

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

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

43

ENGINEERING ARTIFACTS

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

Vision Document

The vision document provides a complete vision for the software system under development and. supports the contract between the funding authority and the development organization. A project vision is meant to be changeable as understanding evolves of the requirements, architecture, plans, and technology. A good vision document should change slowly. Figure 6-9 provides a default outline for a vision document.
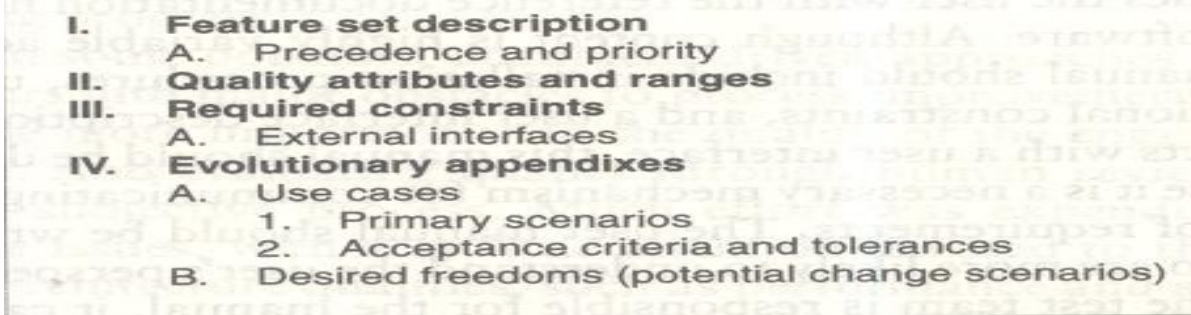
```
I.      Feature set description
        A.    Precedence and priority
II.     Quality attributes and ranges
III.    Required constraints
        A.    External interfaces
IV.     Evolutionary appendixes
        A.    Use cases
              1.    Primary scenarios
              2.    Acceptance criteria and tolerances
        B.    Desired freedoms (potential change scenarios)
```

FIGURE 6-9. *Typical vision document outline*

Architecture Description

The architecture description provides an organized view of the software architecture under development. It is extracted largely from the design model and includes views of the design, implementation, and deployment sets sufficient to understand how the operational concept of the requirements set will be achieved. The breadth of the architecture description will vary from project to project depending on many factors. Figure 6-10 provides a default outline for an architecture description.
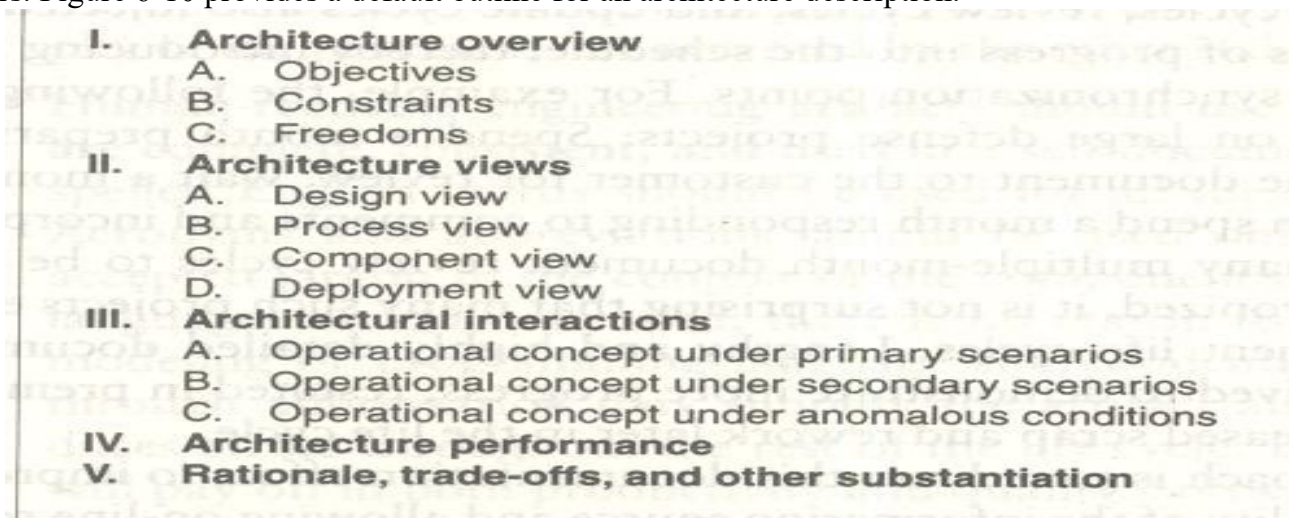
```
I.      Architecture overview
        A.    Objectives
        B.    Constraints
        C.    Freedoms
II.     Architecture views
        A.    Design view
        B.    Process view
        C.    Component view
        D.    Deployment view
III.    Architectural interactions
        A.    Operational concept under primary scenarios
        B.    Operational concept under secondary scenarios
        C.    Operational concept under anomalous conditions
IV.     Architecture performance
V.      Rationale, trade-offs, and other substantiation
```

FIGURE 6-10. *Typical architecture description outline*

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.

PROGRAMMATIC ARTIFACTS

•**People want to review information but don't understand the language of the artifact**. Many interested reviewers of a particular artifact will resist having to learn the engineering language in which the artifact is written. It is not uncommon to find people (such as veteran software managers, veteran quality assurance specialists, or an auditing authority from a regulatory agency) who react as follows: "I'm not going to learn UML, but I want to review the design of this software, so give me a separate description such as some flowcharts and text that I can understand."

•**People want to review the information but don't have access to the tools.** It is not very common for the development organization to be fully tooled; it is extremely rare that the/other stakeholders have any capability to review the engineering artifacts on-line. Consequently, organizations are forced to exchange paper documents. Standardized formats (such as UML, spreadsheets, Visual Basic, C++, and Ada 95), visualization tools, and the Web are rapidly making it economically feasible for all stakeholders to exchange information electronically.

•**Human-readable engineering artifacts should use rigorous notations that are complete, consistent, and used in a self-documenting manner.** Properly spelled English words should be used for all identifiers and descriptions. Acronyms and abbreviations should be used only where they are well accepted jargon in the context of the component's usage. Readability should be emphasized and the use of proper English words should be required in all engineering artifacts. This practice enables understandable representations, browse able formats (paperless review), more-rigorous notations, and reduced error rates.

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

•**Paper is tangible; electronic artifacts are too easy to change**. On-line and Web-based artifacts can be changed easily and are viewed with more skepticism because of their inherent volatility.

Unit – III Important questions

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

Module III: Process
A: Work Flows of the process   **[03 Periods]**
Software process workflows, Iteration workflows.
B: Checkpoints of the process   **[03 Periods]**
Major mile stones, Minor Milestones, Periodic statusassessments.
C: Iterative Process Planning   **[03 Periods]**
Work breakdown structures, planning guidelines, costand schedule estimating, Iteration planning process, Pragmatic planning.

Workflow of the process

SOFTWARE PROCESS WORKFLOWS

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

1. Management workflow: controlling the process and ensuring win conditions for all stakeholders
2. Environment workflow: automating the process and evolving the maintenance environment
3. Requirements workflow: analyzing the problem space and evolving the requirements artifacts
4. Design workflow: modeling the solution and evolving the architecture and design artifacts
5. Implementation workflow: programming the components and evolving the implementation and deployment artifacts
6. Assessment workflow: assessing the trends in process and product quality
7. Deployment workflow: transitioning the end products to the user

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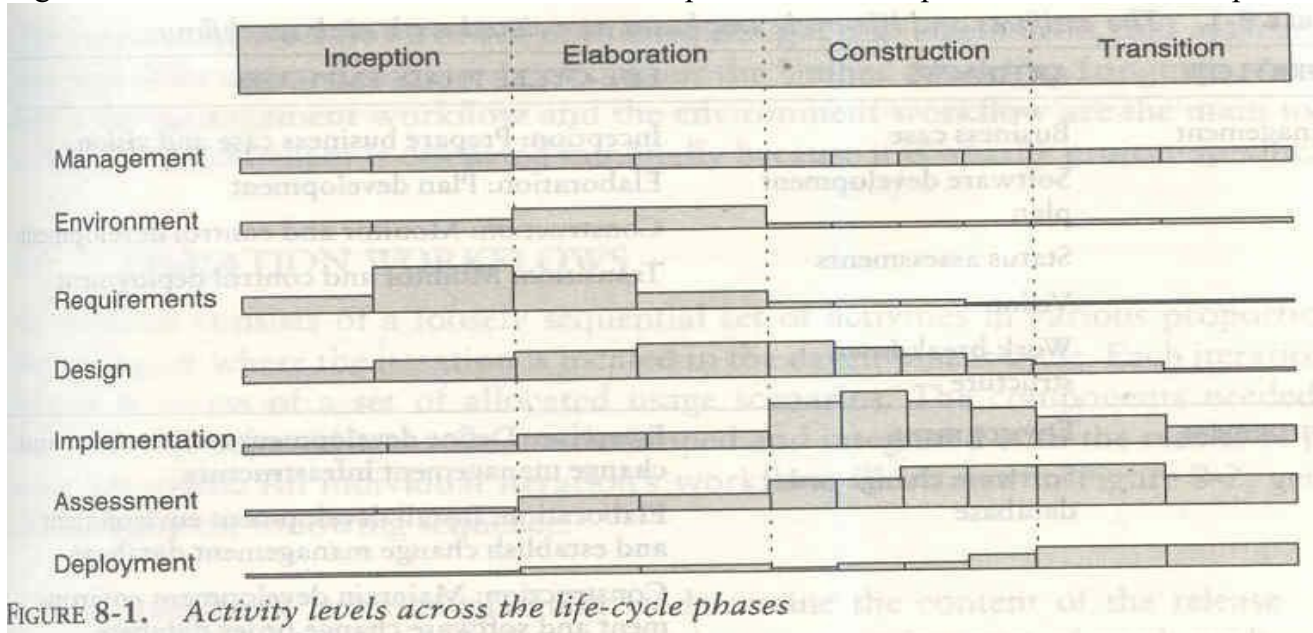

FIGURE 8-1.   *Activity levels across the life-cycle phases*

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

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

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

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

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

- Environment: evolving the software change order database to reflect all new baselines and changes to existing baselines for all product, test, and environment components
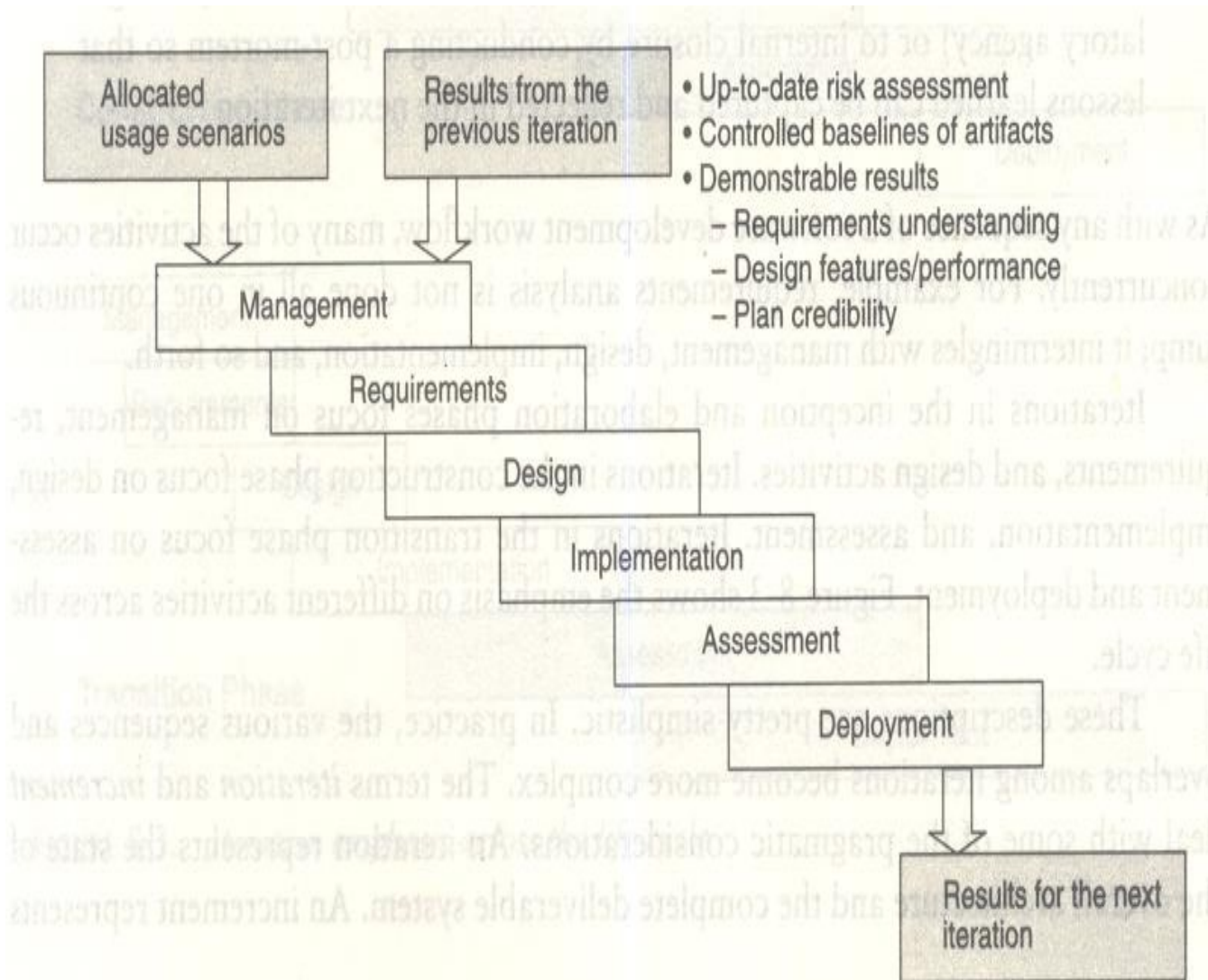
FIGURE 8-2.   *The workflow of an iteration*

- Requirements: analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to be demonstrated at the end of this iteration and their evaluation criteria; updating any requirements set artifacts to reflect changes necessitated by results of this iteration's engineering activities

- Design: evolving the baseline architecture and the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate against the evaluation criteria allocated to this iteration; updating design set artifacts to reflect changes necessitated by the results of this iteration's engineering activities

48

- Implementation: developing or acquiring any new components, and enhancing or modifying any existing components, to demonstrate the evaluation criteria allocated to this iteration; integrating and testing all new and modified components with existing baselines (previous versions)

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

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

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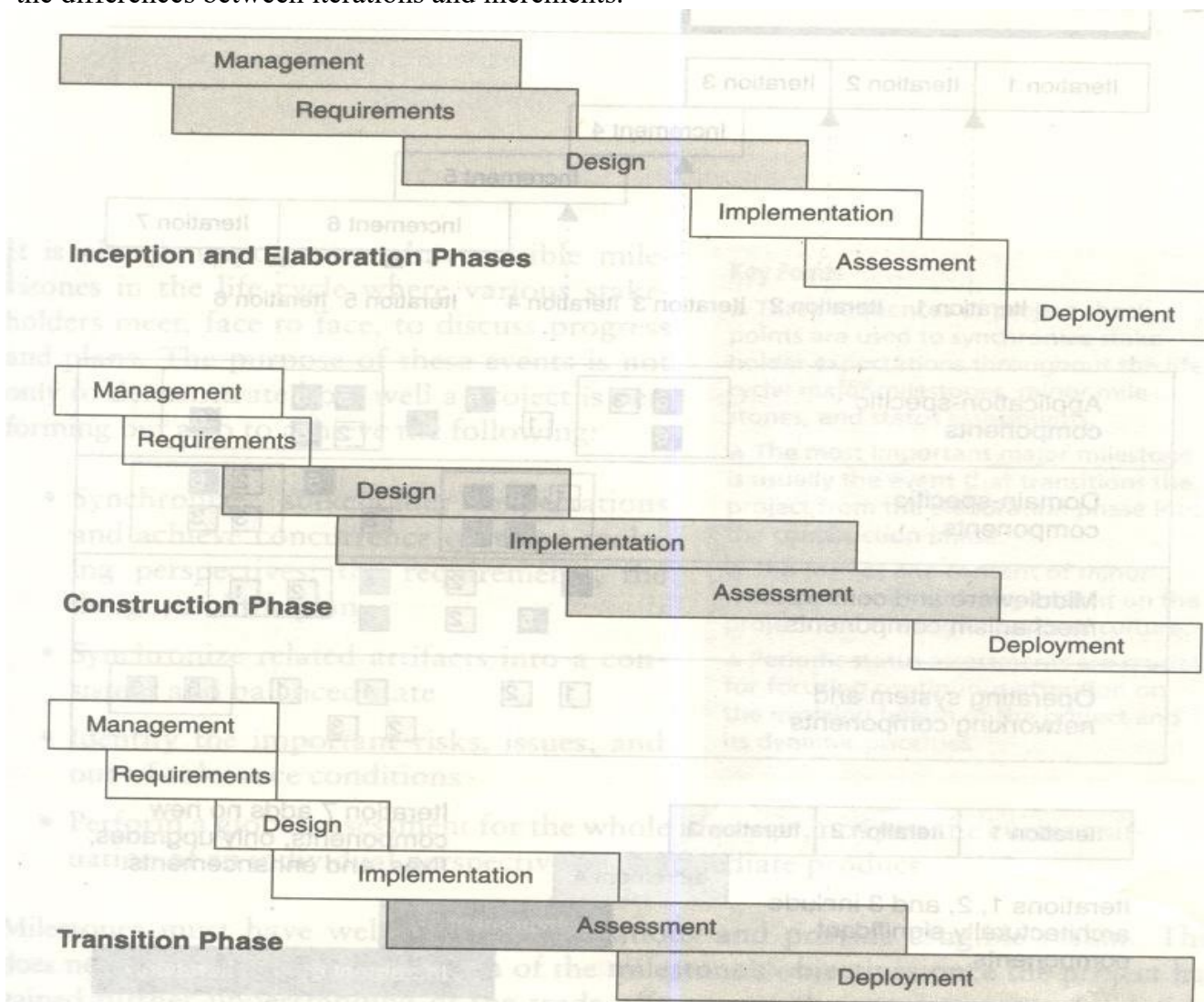
FIGURE 8-3. *Iteration emphasis across the life cycle*

| Inception | Elaboration | Construction | Transition |

Progress

100%

Progress can be measured as the % of components under configuration control, the % of demonstrable use cases, etc.

| Iteration 1 | Iteration 2 | Iteration 3 |
Increment 4
Increment 5
Increment 6 | Iteration 7

Iteration 1   Iteration 2   Iteration 3   Iteration 4   Iteration 5   Iteration 6

Application-specific components

Domain-specific components

Middleware and common mechanism components

Operating system and networking components

| Iteration 1 | Iteration 2 | Iteration 3 |

Iterations 1, 2, and 3 include architecturally significant components.

Increment 4
Increment 5
Increment 6 | Iteration 7

Iteration 7 adds no new components, only upgrades, fixes, and enhancements.
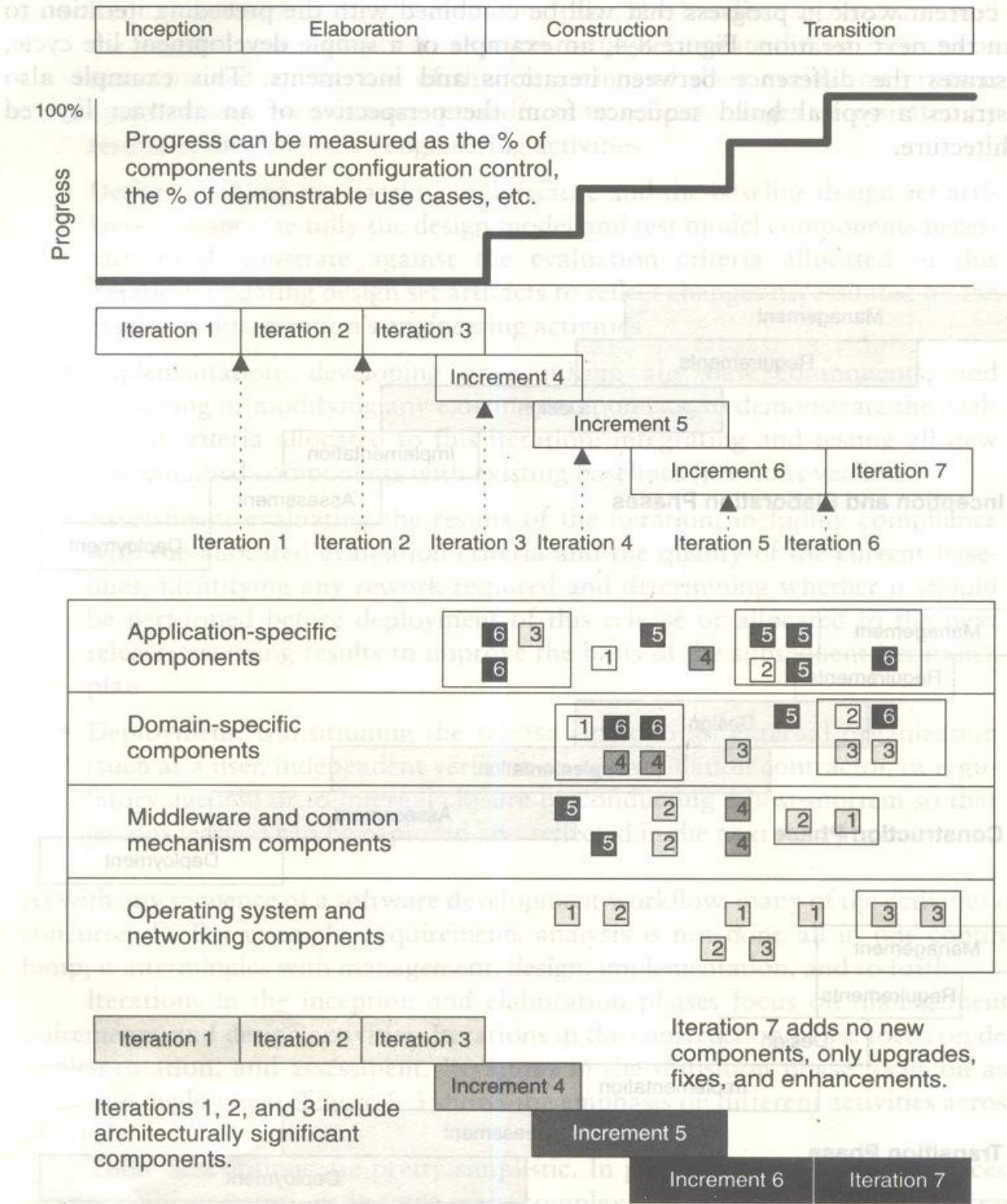
FIGURE 8-4. *A typical build sequence associated with a layered architecture*

9. Checkpoints of the process

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

1. *Major milestones.* These system wide events are held at the end of each development phase. They provide visibility to system wide issues, synchronize the management and engineering perspectives, and verify that the aims of the phase have been achieved.

2. *Minor milestones.* These iteration-focused events are conducted to review the content of an iteration in detail and to authorize continued work.

3. *Status assessments.* These periodic events provide management with frequent and regular insight into the progress being made.

Each of the four phases-inception, elaboration, construction, and transition consists of one or more iterations and concludes with a major milestone when a planned technical capability is produced in demonstrable form. An iteration represents a cycle of activities for which there is a well-defined intermediate result-a minor milestone-captured with two artifacts: a release specification (the evaluation criteria and plan) and a release description (the results). Major milestones at the end of each phase use formal, stakeholder-approved evaluation criteria and release descriptions; minor milestones use informal, development-team-controlled versions of these artifacts.
Figure 9-1 illustrates a typical sequence of project checkpoints for a relatively large project.
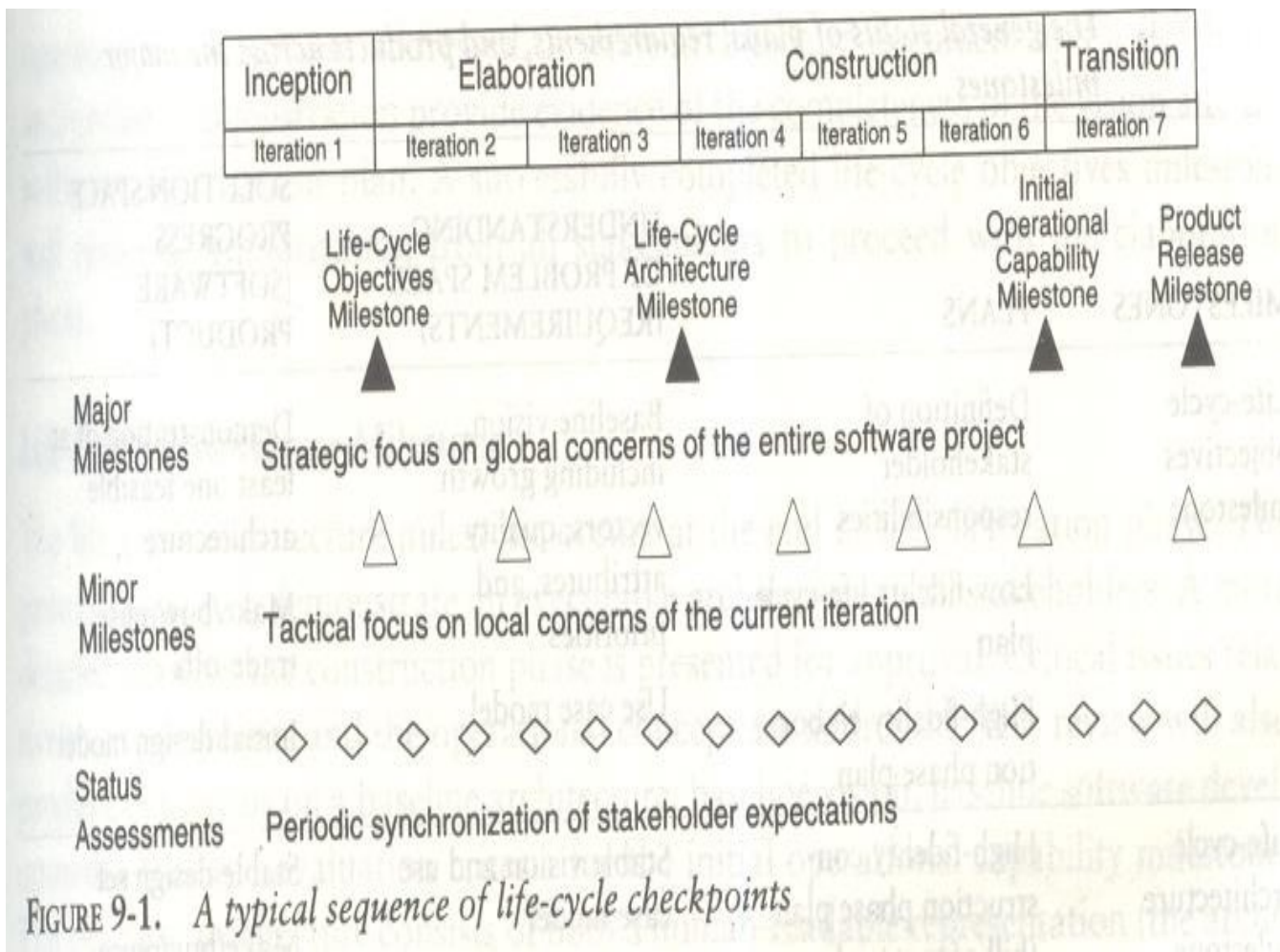
FIGURE 9-1. *A typical sequence of life-cycle checkpoints*

The four major milestones occur at the transition points between life-cycle phases. They can be used in many different process models, including the conventional waterfall model. In an iterative model, the major milestones are used to achieve concurrence among all stakeholders on the current state of the project. Different stakeholders have very different concerns:

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

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

- Architects and systems engineers: product line compatibility, requirements changes, trade-off analyses, completeness and consistency, balance among risk, quality, and usability

- Developers: sufficiency of requirements detail and usage scenario descriptions, . frameworks for component selection or development, resolution of development risk, product line compatibility, sufficiency of the development environment

- Maintainers: sufficiency of product and documentation artifacts, understandability, interoperability with existing systems, sufficiency of maintenance environment

- Others: possibly many other perspectives by stakeholders such as regulatory agencies, independent verification and validation contractors, venture capital investors, subcontractors, associate contractors, and sales and marketing teams

Table 9-1 summarizes the balance of information across the major milestones.

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

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

Life-Cycle Objectives Milestone

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

Life-Cycle Architecture Milestone

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

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

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

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

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

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

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

Initial Operational Capability Milestone

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

Product Release Milestone

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

MINOR MILESTONES

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

- Iteration Readiness Review. This informal milestone is conducted at the start of each iteration to review the detailed iteration plan and the evaluation criteria that have been allocated to this iteration.

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

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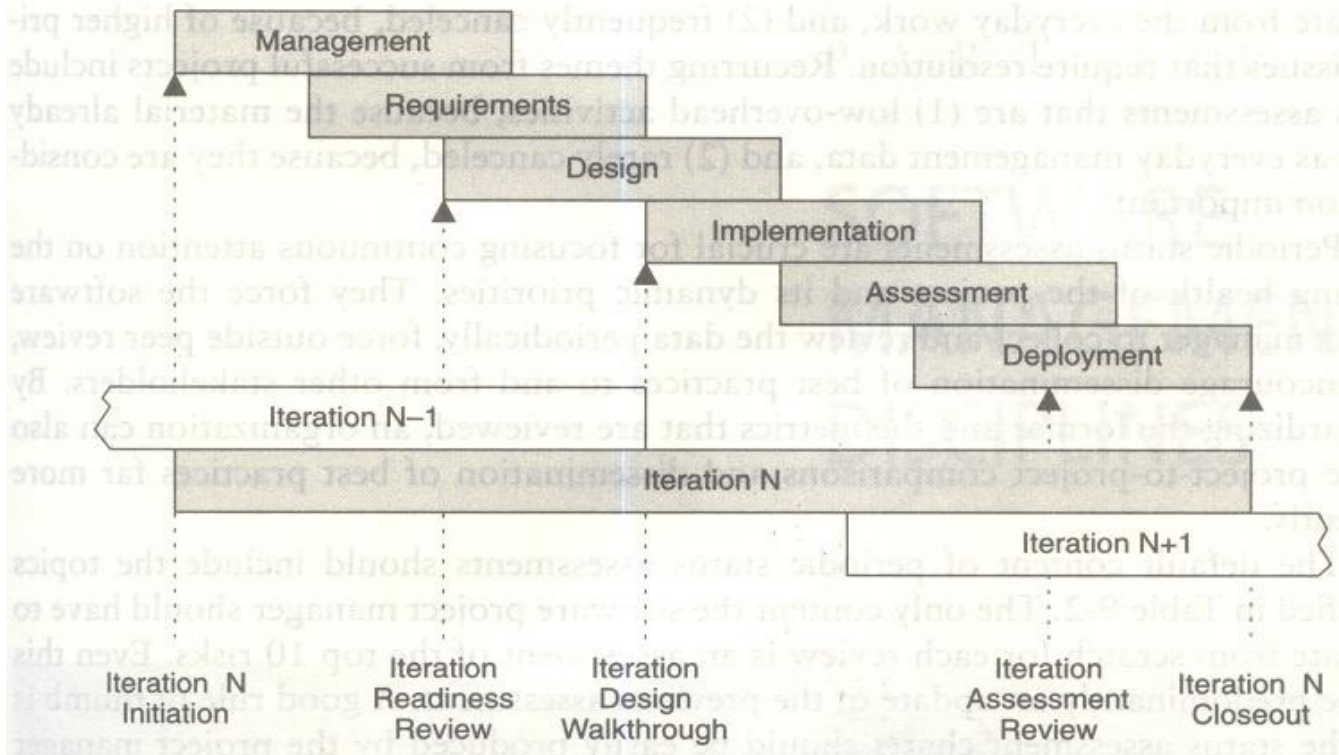

FIGURE 9-4.   *Typical minor milestones in the life cycle of an iteration*

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

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

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

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

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

Periodic status assessments are crucial for focusing continuous attention on the evolving health of the project and its dynamic priorities. They force the software project manager to collect and review the data periodically, force outside peer review, and encourage dissemination of best practices to and from other stakeholders.

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

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

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

10. Iterative process planning

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

A WBS is simply a hierarchy of elements that decomposes the project plan into the discrete work

tasks. A WBS provides the following information structure:

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

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

CONVENTIONAL WBS ISSUES

Conventional work breakdown structures frequently suffer from three fundamental flaws.

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

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

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

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

<u>Figure 10-1 Conventional work breakdown structure, following the product hierarchy</u>

Management
    System requirement and design
    Subsystem 1
    Component 11
    Requirements
    Design
    Code
    Test
    Documentation
    …(similar structures for other components)
    Component 1N
    Requirements
    Design
    Code
    Test
    Documentation
    …(similar structures for other subsystems)
    Subsystem M
    Component M1

Requirements
Design
Code
Test
Documentation
…(similar structures for other components)
Component MN
Requirements
Design
Code
Test
Documentation
Integration and test
Test planning
Test procedure preparation
Testing
Test reports
Other support areas
Configuration control
Quality assurance
System administration

EVOLUTIONARY WORK BREAKDOWN STRUCTURES

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

- First-level WBS elements are the workflows (management, environment, requirements, design, implementation, assessment, and deployment).

- Second-level elements are defined for each phase of the life cycle (inception, elaboration, construction, and transition).

- Third-level elements are defined for the focus of activities that produce the artifacts of each phase.

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

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

- Scale. Larger projects will have more levels and substructures.

- Organizational structure. Projects that include subcontractors or span multiple organizational entities may introduce constraints that necessitate different WBS allocations.
- Degree of custom development. Depending on the character of the project, there can be very different emphases in the requirements, design, and implementation workflows.
- Business context. Projects developing commercial products for delivery to a broad customer base may require much more elaborate substructures for the deployment element.

- Precedent experience. Very few projects start with a clean slate. Most of them are developed as new generations of a legacy system (with a mature WBS) or in the context of existing organizational standards (with preordained WBS expectations).

The WBS decomposes the character of the project and maps it to the life cycle, the budget, and the

personnel. Reviewing a WBS provides insight into the important attributes, priorities, and structure of the project plan.

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

Figure 10-2 Default work breakdown structure

A   Management

  AA Inception phase management

      AAA   Business case development

      AAB   Elaboration phase release specifications

      AAC   Elaboration phase WBS specifications

      AAD   Software development plan

      AAE   Inception phase project control and status assessments

  AB Elaboration phase management

      ABA   Construction phase release specifications

      ABB   Construction phase WBS baselining

      ABC   Elaboration phase project control and status assessments

  AC   Construction phase management

      ACA   Deployment phase planning

      ACB   Deployment phase WBS baselining

      ACC   Construction phase project control and status assessments

  AD   Transition phase management

      ADA   Next generation planning

      ADB   Transition phase project control and status assessments

B   Environment

  BA   Inception phase environment specification

  BB   Elaboration phase environment baselining

      BBA   Development environment installation and administration

      BBB   Development environment integration and custom toolsmithing

      BBC   SCO database formulation

  BC   Construction phase environment maintenance

      BCA   Development environment installation and administration

      BCB   SCO database maintenance

  BD   Transition phase environment maintenance

      BDA   Development environment maintenance and administration

      BDB   SCO database maintenance

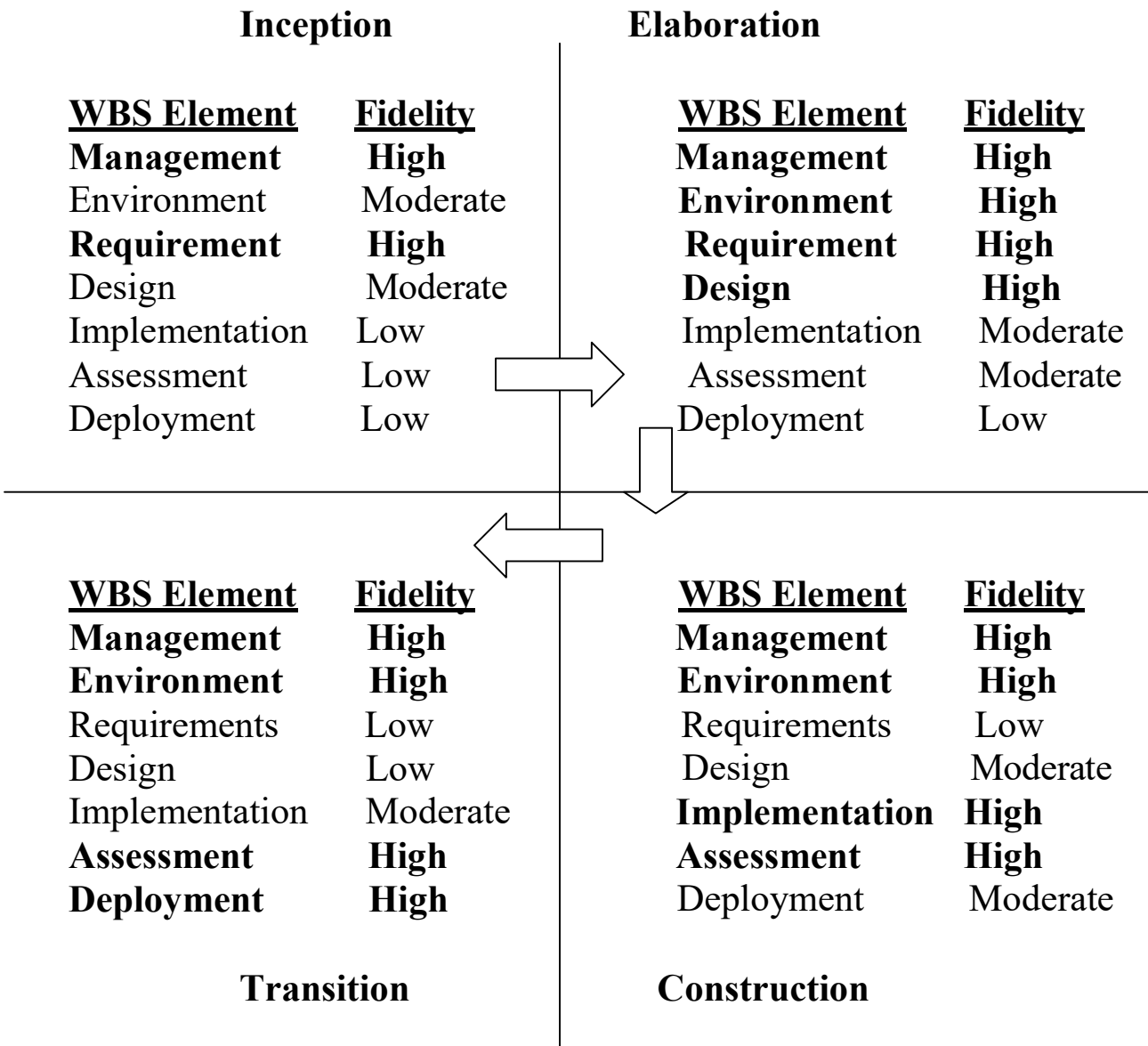   **BDC   Maintenance environment packaging and transition**

C   Requirements

  CA   Inception phase requirements development

      CCA   Vision specification

      CAB   Use case modeling

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

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

Inception

WBS Element	Fidelity
Management	**High**
Environment	Moderate
Requirement	**High**
Design	Moderate
Implementation	Low
Assessment	Low
Deployment	Low

Elaboration

WBS Element	Fidelity
Management	**High**
Environment	**High**
Requirement	**High**
Design	**High**
Implementation	Moderate
Assessment	Moderate
Deployment	Low

WBS Element	Fidelity
Management	**High**
Environment	**High**
Requirements	Low
Design	Low
Implementation	Moderate
Assessment	**High**
Deployment	**High**

WBS Element	Fidelity
Management	**High**
Environment	**High**
Requirements	Low
Design	Moderate
Implementation	**High**
Assessment	**High**
Deployment	Moderate

Transition   Construction

PLANNING GUIDELINES

Software projects span a broad range of application domains. It is valuable but risky to make specific planning recommendations independent of project context. Project-independent planning advice is also risky. There is the risk that the guidelines may pe adopted blindly without being adapted to specific project circumstances. Two simple planning guidelines should be considered when a project plan is being initiated or assessed. The first guideline, detailed in Table 10-1, prescribes a default allocation of costs among the first-level WBS elements. The second guideline, detailed in Table 10-2, prescribes the allocation of effort and schedule across the lifecycle phases.

Web budgeting defaults

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

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

Domain	Inception	Elaboration	Construction	Transition
Effort	5%	20%	65%	10%
Schedule	10%	30%	50%	10%

THE COST AND SCHEDULE ESTIMATING PROCESS

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

1. The software project manager (and others) develops a characterization of the overall size, process, environment, people, and quality required for the project.

2. A macro-level estimate of the total effort and schedule is developed using a software cost estimation model.

3. The software project manager partitions the estimate for the effort into a top-level WBS using guidelines such as those in Table 10-1.

4. At this point, subproject managers are given the responsibility for decomposing each of the WBS elements into lower levels using their top-level allocation, staffing profile, and major milestone dates as constraints.

The second perspective is a backward-looking, bottom-up approach. We start with the end in mind, analyze the micro-level budgets and schedules, then sum all these elements into the higher level budgets and intermediate milestones. This approach tends to define and populate the WBS from the lowest levels upward. From this per- spective, the following planning sequence would occur:
1. The lowest level WBS elements are elaborated into detailed tasks

2. Estimates are combined and integrated into higher level budgets and milestones.

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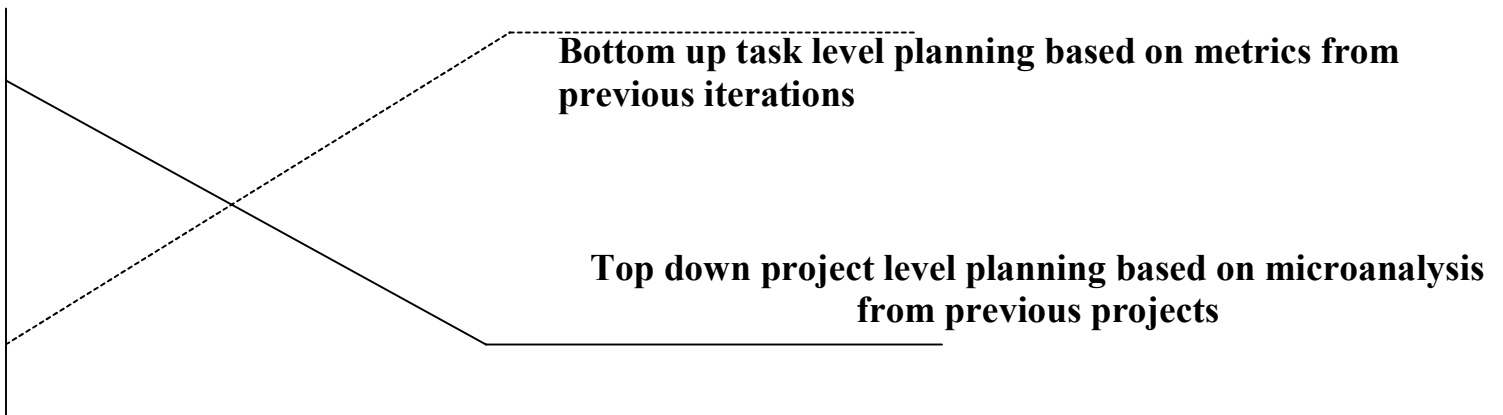
Milestone scheduling or budget allocation through top-down estimating tends to exaggerate the project management biases and usually results in an overly optimistic plan. Bottom-up estimates usually exaggerate the performer biases and result in an overly pessimistic plan.

These two planning approaches should be used together, in balance, throughout the life cycle of the project. During the engineering stage, the top-down perspective will dominate because there is usually not enough depth of understanding nor stability in the detailed task sequences to perform credible bottom-up planning. During the production stage, there should be enough precedent experience and planning fidelity

that the bottom-up planning perspective will dominate. Top-down approach should be well tuned to the project-

specific parameters, so it should be used more as a global assessment technique. Figure 10-4 illustrates this life- cycle planning balance.

Figure 10-4 Planning balance throughout the life cycle

Bottom up task level planning based on metrics from previous iterations

Top down project level planning based on microanalysis from previous projects

Engineering Stage		Production Stage	
Inception	Elaboration	Construction	Transition

Feasibility iteration   Architecture iteration   Usable iteration   Product Releases

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

THE ITERATION PLANNING PROCESS

Planning is concerned with defining the actual sequence of intermediate results. An evolutionary build plan is important because there are always adjustments in build content and schedule as early conjecture evolves into well-understood project circumstances. *Iteration* is used to mean a complete synchronization across the project, with a well-orchestrated global assessment of the entire project baseline.

- Inception iterations. The early prototyping activities integrate the foundation components of a candidate architecture and provide an executable framework for elaborating the critical use cases of the system. This framework includes existing components, commercial components, and custom prototypes sufficient to demonstrate a candidate architecture and sufficient requirements understanding to establish a credible business case, vision, and software development plan.
- Elaboration iterations. These iterations result in architecture, including a complete framework and

infrastructure for execution. Upon completion of the architecture iteration, a few critical use cases should

be demonstrable: (1) initializing the architecture, (2) injecting a scenario to drive the worst-case data processing flow through the system (for example, the peak transaction throughput or peak load scenario), and (3) injecting a scenario to drive the worst-case control flow through the system (for example, orchestrating the fault-tolerance use cases).

- Construction iterations. Most projects require at least two major construction iterations: an alpha release and a beta release.
- Transition iterations. Most projects use a single iteration to transition a beta release into the final product.

The general guideline is that most projects will use between four and nine iterations. The typical project would have the following six-iteration profile:

- One iteration in inception: an architecture prototype

- Two iterations in elaboration: architecture prototype and architecture baseline

- Two iterations in construction: alpha and beta releases

- One iteration in transition: product release

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

PRAGMATIC PLANNING

Even though good planning is more dynamic in an iterative process, doing it accurately is far easier. While executing iteration N of any phase, the software project manager must be monitoring and controlling against a plan that was initiated in iteration N - 1 and must be planning iteration N + 1. The art of good project· management is to make trade-offs in the current iteration plan and the next iteration plan based on objective results in the current iteration and previous iterations. Aside from bad architectures and misunderstood requirements, inadequate planning (and subsequent bad management) is one of the most common reasons for project failures. Conversely, the success of every successful project can be attributed in part to good planning.

A project's plan is a definition of how the project requirements will be transformed into' a product within the business constraints. It must be realistic, it must be current, it must be a team product, it must be understood by the stakeholders, and it must be used. Plans are not just for managers. The more open and visible the planning process and results, the more ownership there is among the team members who need to execute it. Bad, closely held plans cause attrition. Good, open plans can shape cultures and encourage teamwork.

Unit – Important Questions

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

Module IV: Automation
A: Process Automation   **[02 Periods]**
Automation Building blocks.
B: Project Control and Process instrumentation   **[05 Periods]**
The seven core Metrics, Management indicators, quality indicators, life cycle expectations, pragmatic
Software Metrics, Metrics automation.
C: Tailoring the Process   **[02 Periods]**
Process discriminates.

═══

A: Process Automation
Many software development organizations are focused on evolving mature processes to improve the
predictability of software management and the performance of their software lines of business (in terms of
product quality, time to market, return on investment, and productivity). While process definition and
tailoring are necessary, a significant level of process automation is also required in order for modern
software development projects to operate profitably.

Automation needs grow depending on the scale of the effort. Just as the construction process varies
depending on whether you are building a dollhouse, a single-family home, or a skyscraper, the software
process varies across the spectrum from single-person spreadsheet tasks to large-scale, multiple-
organization, catastrophic cost-of-failure applications. The techniques, training, time scales, acceptance
criteria, and levels of automation differ significantly at opposite ends of the spectrum.

Most software organizations are confronted with the task of integrating their own environment and
infrastructure for software development. This process typically results in the selection of more or less
incompatible tools that have different information repositories, are supplied by different vendors, work on
different platforms, use different jargon, and are based on different process assumptions. Integrating such
an infrastructure has proven to be much more problematic than expected.

Key Points a The environment must be a first-class artifact of the process.
▲ Process automation, and change management in particular, are critical to an iterative process. If change
is too expensive, the development organization will resist it.
▲ Round-trip engineering and integrated environments promote change freedom and effective evolution
of technical artifacts.
▲ Metrics automation is crucial to effective project control.
▲ External stakeholders need access to environment resources to improve interaction with the
development team and add value to the process.
Automating the development process and establishing an infrastructure for supporting the various project
workflows are important activities of the engineering stage of the life cycle. They include the tool
selection, custom toolsmithing, and process automation necessary to perform against the development
plan with acceptable efficiency. Evolving the development environment into the maintenance environment
is also crucial to any long-lived software development project.

To complicate matters further, it is rare to find stakeholders who treat the environment as a first-class
artifact necessary for the continued maintenance of the product. The environment that provides the process
automation is a tangible artifact that is critical to the life-cycle cost of the system being developed. The
top-level WBS recommended in Chapter 10 recognizes the environment as a first-class workflow.

Introduced three levels of process. Each level requires a certain degree of process automation for the
corresponding process to be carried out efficiently:
1. **Meta process:** an organization's policies, procedures, and practices for managing a software-intensive
line of business. The automation support for this level is called an infrastructure. An infrastructure is an
inventory of preferred tools, artifact templates, microprocess guidelines, macroprocess guidelines, project
performance repository, database of organizational skill sets, and library of precedent examples of past

project plans and results.

2. **Macro process:** a project's policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality constraints. The automation support for a project's process is called an environment. An environment is a specific collection of tools to produce a specific set of artifacts as governed by a specific project plan.

3. **Micro process:** a project team's policies, procedures, and practices for achieving an artifact of the software process. The automation support for generating an artifact is generally called a tool. Typical tools include requirements management, visual modeling, compilers, editors, debuggers, change management, metrics automation, document automation, test automation, cost estimation, and workflow automation.

Tools: Automation Building blocks:

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

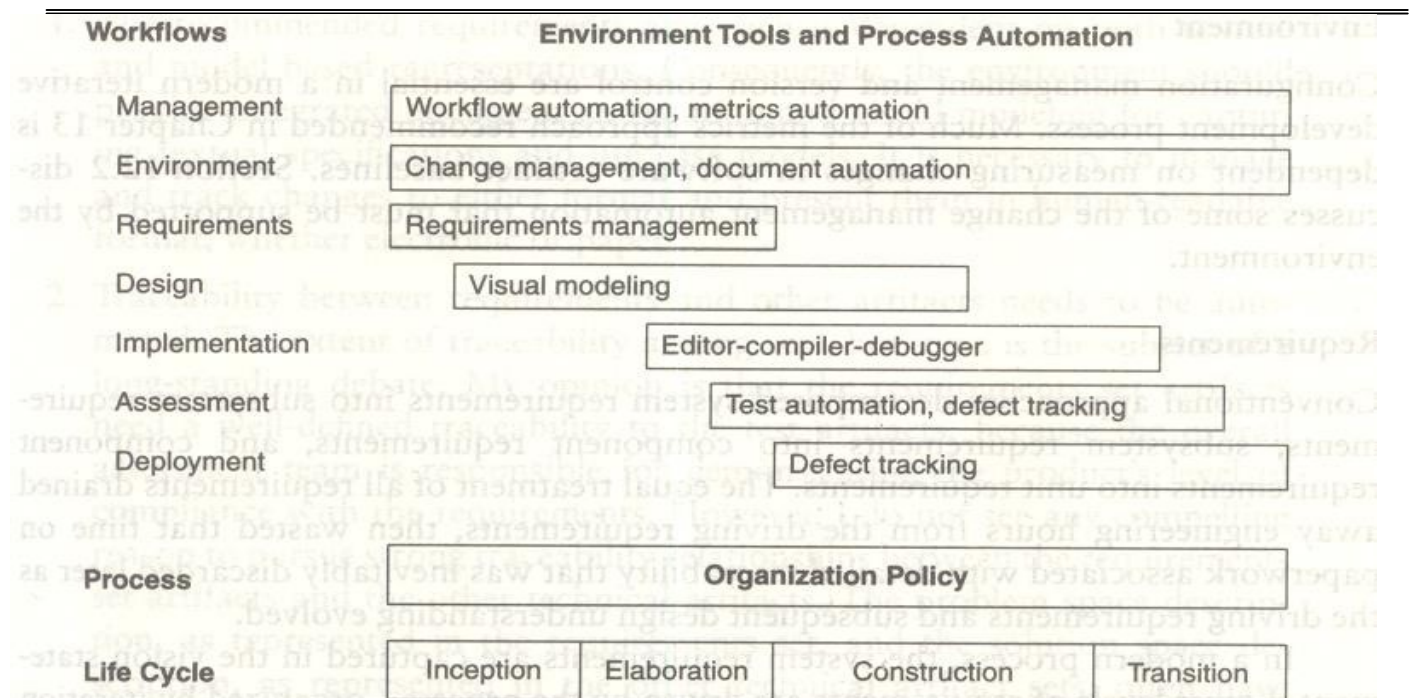| Workflows	Environment Tools & process Automation
Management automation	Workflow automation, Metrics
Environment	Change Management, Document Automation
Requirements	Requirement Management
Design	Visual Modeling
Implementation	-Editors, Compilers, Debugger, Linker, Runtime
Assessment	-Test automation, defect Tracking
Deployment	defect Tracking

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

B: Project Control and Process instrumentation

PROJECT CONTROL & PROCESS INSTRUMENTATION

INTERODUCTION: Software metrics are used to implement the activities and products of the

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

Need for Software Metrics:

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

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

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

INDICATORS:

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

Two types of indicators are:

(i) Management indicators.

(ii) Quality indicators.

Management Indicators

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

Quality Indicators

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

SEVEN CORE METRICS OF SOFTWARE PROJECT

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

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

Seven core metrics related to project control:

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

MANAGEMENT INDICATORS:

Work and progress

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

The default perspectives of this metric are:

Software architecture team: - Use cases demonstrated.

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

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

Software management team: - milestones completed.

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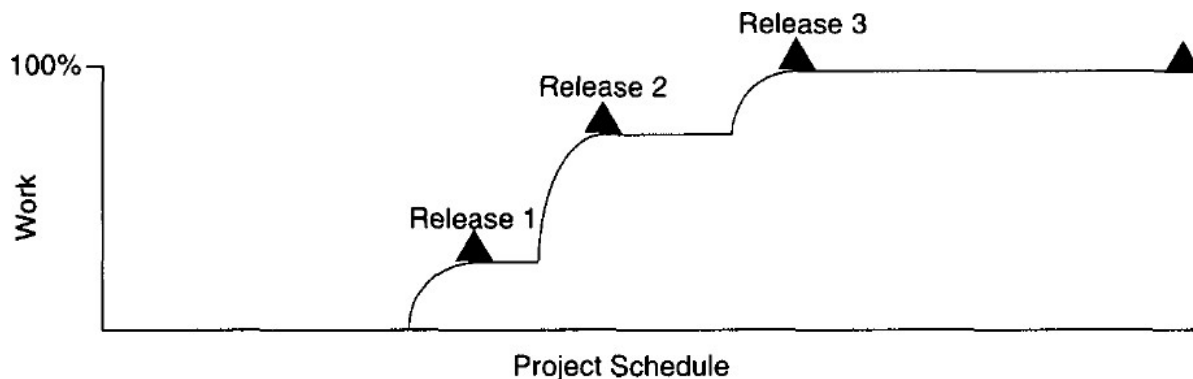

Fig: work and progress

Budgeted cost and expenditures

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

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

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

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

schedule. **Earned value**: It is the value that represents the planned cost of the

actual progress. **Cost variance**: It is the difference between the actual cost and

the earned value.

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

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

subjective

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

Staffing and team dynamics

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

These three management indicators are responsible for technical progress, financial status and staffing progress.
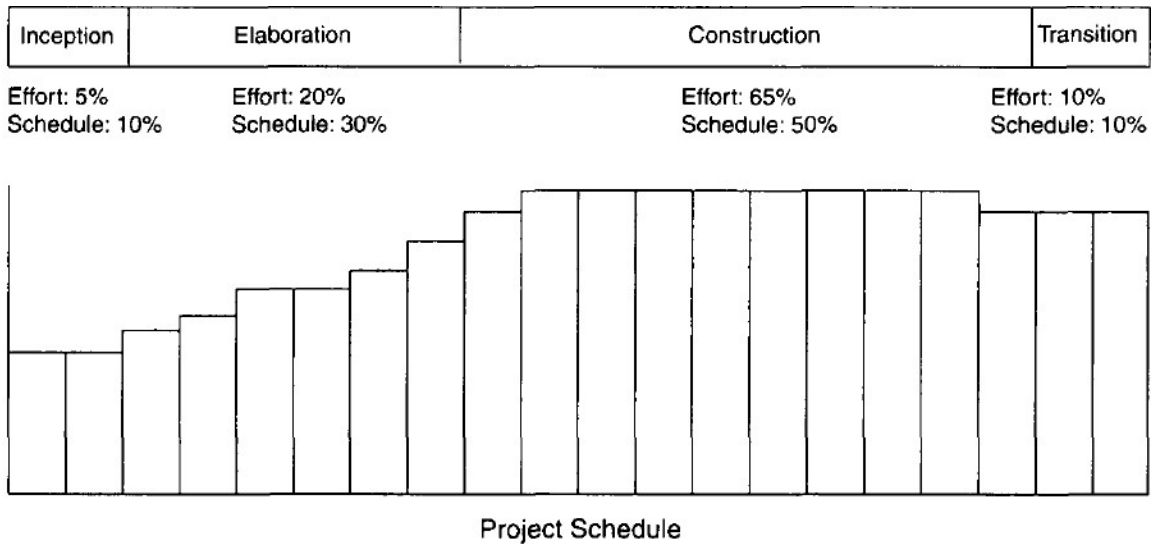
. *Fig: work and progress*

Fig: staffing and Team dynamics

QUALITY INDICATORS:

Change traffic and stability:

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

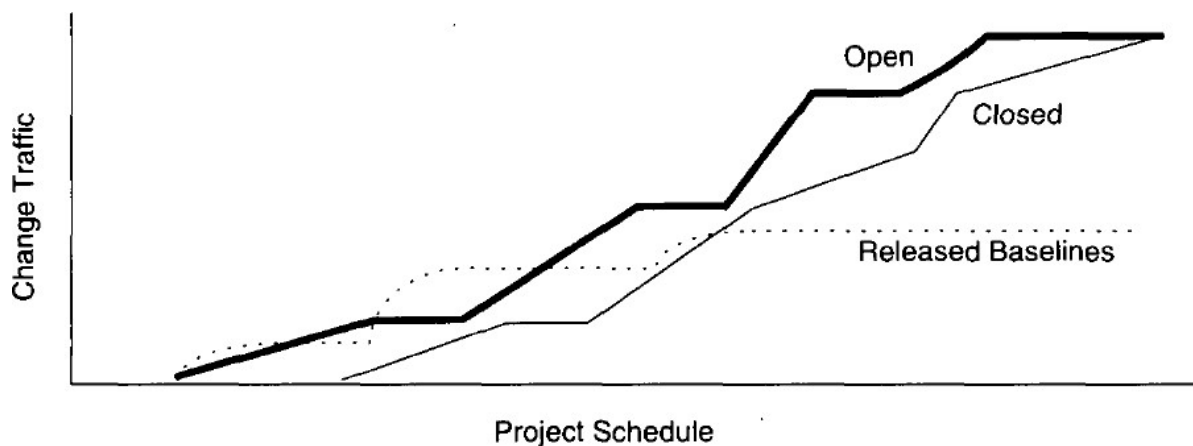## *The below figure shows stability expectation over a healthy project's life cycle*

Fig: Change traffic and stability

Breakage and modularity   72

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 rewoke SLOC per change, by change type, by release, by components and by subsystems.

Rework and adaptability:

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

MTBF **and Maturity:**

This metric measures defect 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.

Lifecycle 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

Pragmatic Software Metrics

Measuring is useful, but it doesn't do any thinking for the decision makers. It only provides data to help them ask-tne right questions, understand the context, and make objective decisions. Because of the highly dynamic nature of software projects, these measures must be available at any time, tailorable to various subsets of the evolving product (release, version, component, class), and maintained so that trends can be assessed (first and second derivatives with respect to time). This situation has been achieved in practice only in projects where the metrics were maintained on-line as an automated by-product of the development/integration environment. The basic characteristics of a good metric are as follows:

1. It is considered meaningful by the customer, manager, and performer. If any one of these stakeholders does not see the metric as meaningful, it will not be used. "The customer is always right" is a sales motto, not an engineering tenet. Customers come to software engineering providers because the providers are more expert than they are at developing and managing software. Customers will accept metrics that are demonstrated to be meaningful to the developer.

2. It demonstrates quantifiable correlation between process perturbations and business performance. The only real organizational goals and objectives are financial: cost reduction, revenue increase, and margin increase.

3. It is objective and unambiguously defined. Objectivity should translate into some form of numeric representation (such as numbers, percentages, ratios) as opposed to textual representations (such as excellent, good, fair, poor). Ambiguity is minimized through well-understood units of measurement (such as staff-month, SLOC, change, function point, class, scenario, requirement), which are surprisingly

hard to define precisely in the software engineering world.

4. It displays trends. This is an important characteristic. Understanding the change in a metric's value with respect to time, subsequent projects, subsequent releases, and so forth is an extremely important perspective, especially for today's iterative development models. It is very rare that a given metric drives the appropriate action directly. More typically, a metric presents a perspective. It is up to the decision authority (manager, team, or other information processing entity) to interpret the metric and decide what action is necessary.

5. It is a natural by-product of the process. The metric does not introduce new artifacts or overhead activities; it is derived directly from the mainstream engineering and management workflows.

6. It is supported by automation. Experience has demonstrated that the most successful metrics are those that are collected and reported by automated tools, in part because software tools require rigorous definitions of the data they process.

When metrics expose a problem, it is important to get underneath all the symptoms and diagnose it. Metrics usually display effects; the causes require synthesis of multiple perspectives and reasoning. For example, reasoning is still required to interpret the following situations correctly:

• A low number of change requests to a software baseline may mean that the software is mature and error-free, or it may mean that the test team is on vacation.

• A software change order that has been open for a long time may mean that the problem was simple to diagnose and the solution required substantial rework, or it may mean that a problem was very time-consuming to diagnose and the solution required a simple change to a single line of code.

• A large increase in personnel in a given month may cause progress to increase proportionally if they are trained people who are productive from the outset. It may cause progress to decelerate if they are untrained new hires who demand extensive support from productive people to get up to speed.

Value judgments cannot be made by metrics; they must be left to smarter entities such as software project managers.

METRICS AUTOMATION:

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

SPCP:

To implement a complete SPCP, the following are necessary.

➢ Metrics primitives - trends, comparisons and progressions

➢ A graphical user interface.

➢ Metrics collection agents

➢ Metrics data management server

➢ Metrics definitions - actual metrics presentations for requirements progress, implementation progress, assessment progress, design progress and other progress dimensions.

➢ Actors - monitor and administrator.

Monitor defines panel layouts, graphical objects and linkages to project data.

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

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

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

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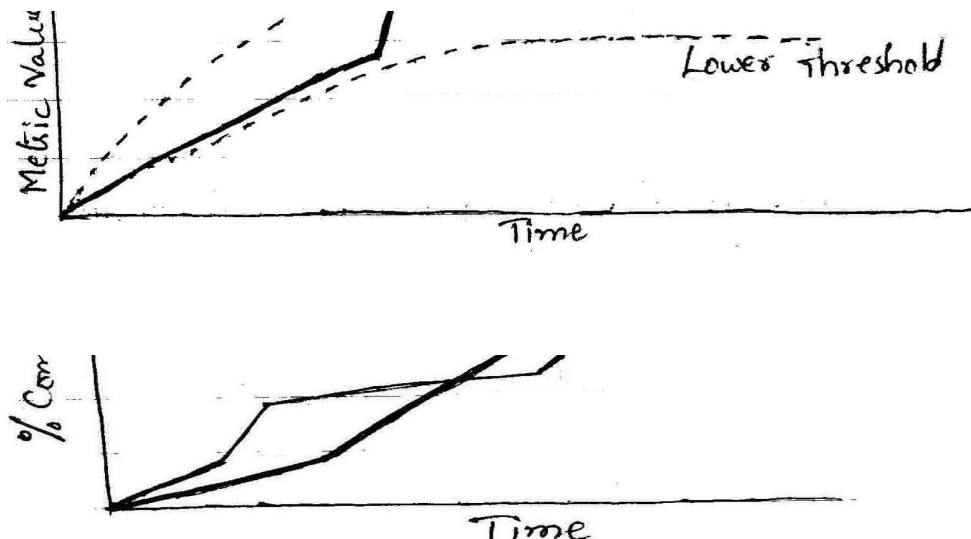
The format and content of any project panel are configurable to the software project manager's preference for tracking metrics of top-level interest. The basic operation of an SPCP can be described by the following top -level use case.

i. Start the SPCP

ii. Select a panel preference

iii. Select a value or graph metric

iv. Select to superimpose controls

v. Drill down to trend

vi. Drill down to point in time.

vii. Drill down to lower levels of information

viii. Drill down to lower level of indicators.

C: Tailoring the Process

Software management efforts span a broad range of domains. While there are some universal themes and techniques, it is always necessary to tailor the process to the specific needs of the project at hand. A commercial software tool developer with complete control of its investment profile will use a very different process from that of a software integrator on contract to automate the security system for a nuclear [power plant](#). There is no doubt that a mature process and effective software management approaches offer much greater value to the large-scale software integrator than they do to the small-scale tool developer. Nevertheless, relative to their business goals, the return on investment realized by better software management approaches is worthwhile for any software organization.

Key Points

▲ The process framework must be configured to the specific characteristics of the project.

▲ The scale of the project—in particular, team size—drives the process configuration more than any other factor.

▲ Other key factors include stakeholder relationships, process flexibility, process maturity, architectural risk, and domain experience.

AWhile specific process implementations will vary, the spirit underlying the process is the same.

Process Discriminants:

In tailoring the management process to a specific domain or project, there are two dimensions of discriminating factors: technical complexity and management com-pJexity. Figure 14-1 illustrates these two dimensions of process variability and shows some example project applications. The formality of reviews, the quality control of artifacts, the priorities of concerns, and numerous other process instantiation parameters are governed by the point a project occupies in these two dimensions. Figure 14-2 summarizes the different priorities along the two dimensions.

Average software project-. 5 to 10 people 10 to 12 months 3 to 5 external interfaces Some unknowns, risks

Lower

Management Complexity

• Smaller scale

• Few stakeholders

Module V: Responsibilities and Case Study
A: Project Organizations and Responsibilities **[04 Periods]**
Line-of-Business Organizations, Understanding Behavior – Organizational Behavior.
B: Future Software Project Management **[03 Periods]**
Modern Project Profiles, Next generationSoftware economics, modern process transitions.
C: Case Study **[02 Periods]**
The command Center Processing and Display system-Replacement(CCPDS-R)

Project Organizations and Responsibilities:

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

1) Line-Of-Business Organizations:
The main features of default organization are as follows:
- Responsibility for process definition & maintenance is specific to a cohesive line of business.
- Responsibility for process automation is an organizational role & is equal in importance to the process definition role.
- Organizational role may be fulfilled by a single individual or several different teams.
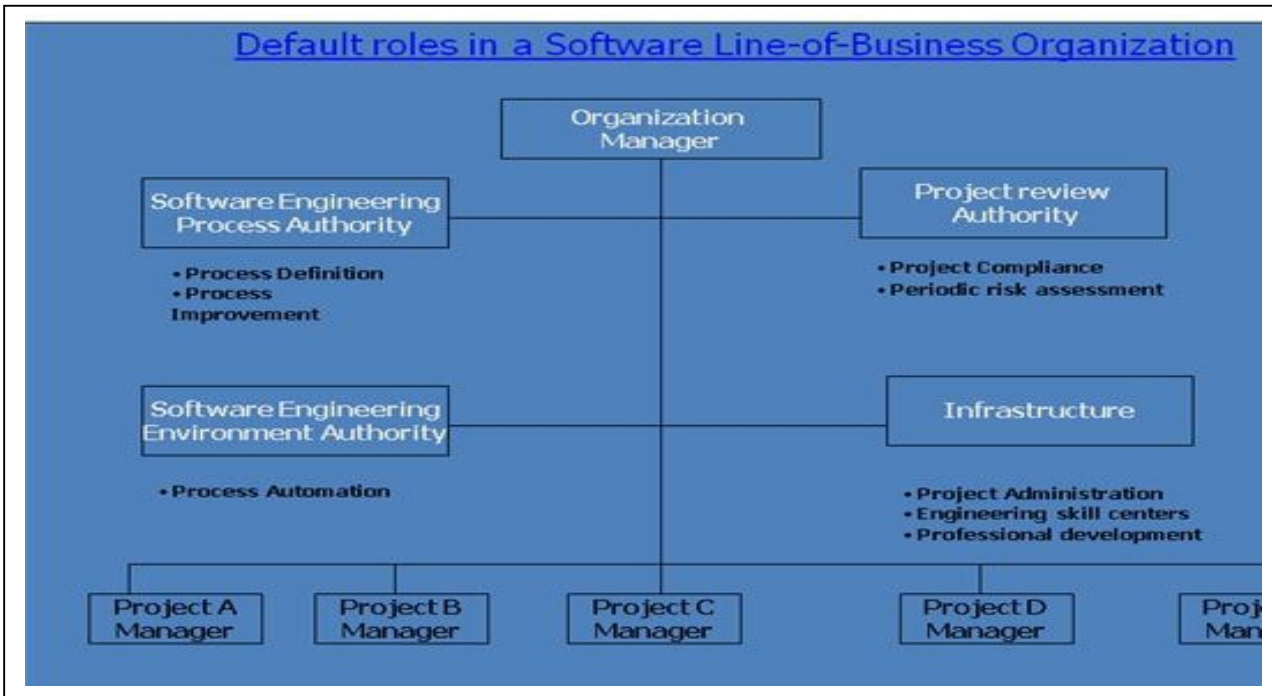
Fig: Default roles in a software Line-of-Business Organization.

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

This role is accountable to General Manager for maintaining a current
assessment of the organization's process maturity & its plan for future
improvement
Project Review Authority (PRA)
The PRA is the single individual responsible for ensuring that a
software project complies with all organizational & business unit
software policies, practices & standards

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

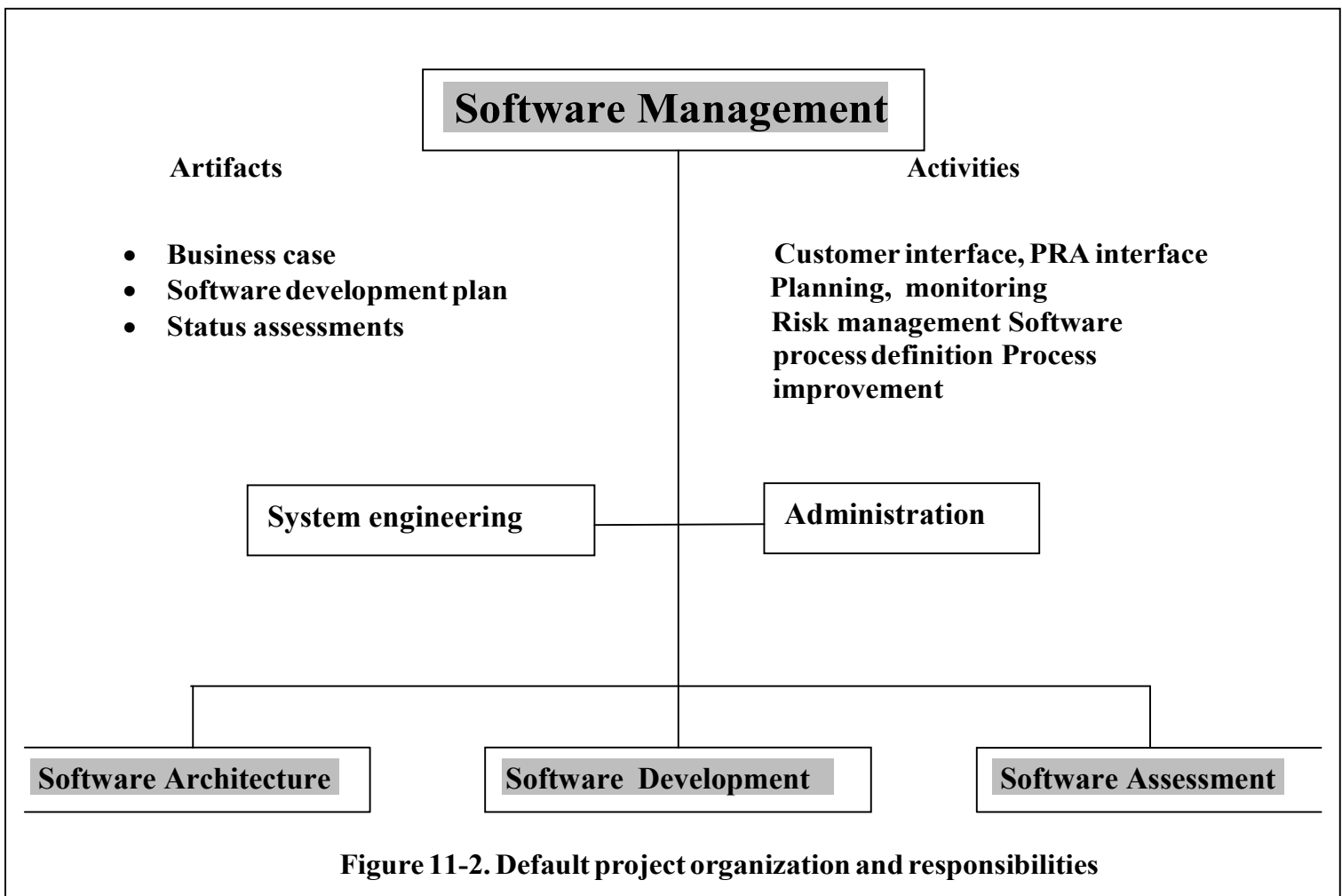**Software Engineering Environment Authority(SEEA)**

The SEEA is responsible for automating the organization's process,
maintaining the organization's standard environment, Training
projects to use the environment & maintaining organization-wide
reusable assets
The SEEA role is necessary to achieve a significant ROI for common process.

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

2) Project organizations:

Software Management

Artifacts

- **Business case**
- **Software development plan**
- **Status assessments**

Activities

**Customer interface, PRA interface
Planning, monitoring
Risk management Software
process definition Process
improvement**

System engineering

Administration

Software Architecture

Software Development

Software Assessment

Figure 11-2. Default project organization and responsibilities

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

3) EVOLUTION OF ORGANIZATIONS:

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

• Inception team: an organization focused on planning, with enough support from the other teams to ensure that the plans represent a consensus of all perspectives
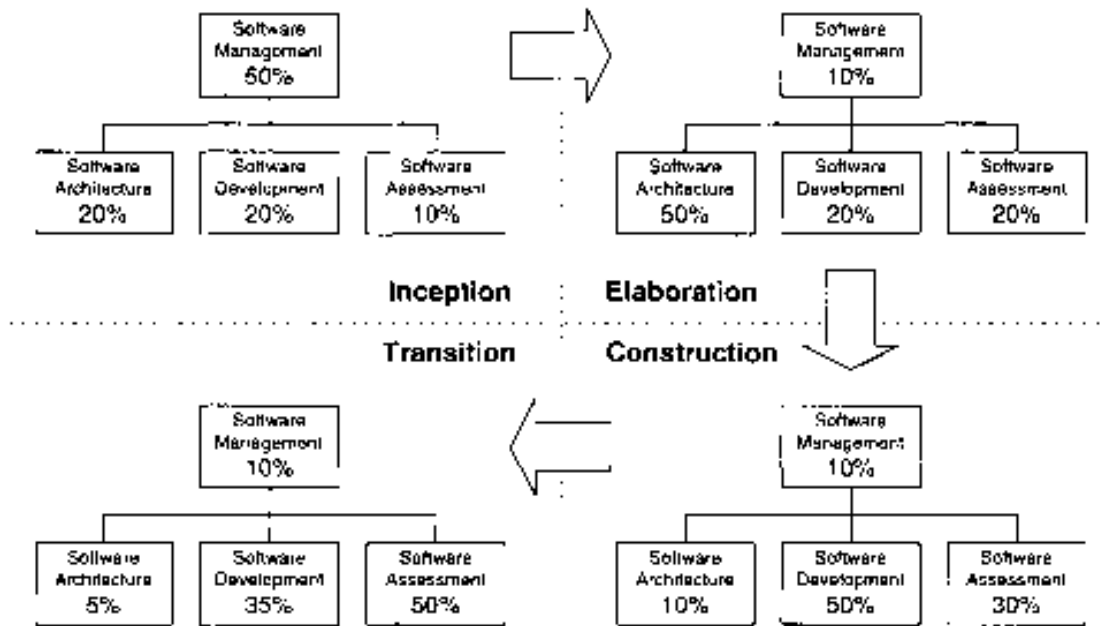
80

Figure 11-7. 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 and software assessment teams as necessary to achieve a stable architecture baseline

• Construction team: a fairly balanced organization in which most of the activity resides in the software development and software assessment teams

• Transition team: a customer-focused organization in which usage feedback drives the deployment activities

It is equally important to elaborate the details of subteams, responsibilities, and work packages, but not until the planning details in the WBS are stable. Defining all the details of lower level team structures prematurely can result in serious downstream inefficiencies.