

RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Computer Science and Engineering, VI-Semester

Open Elective - CS604 (A) Knowledge Management

OBJECTIVES: The student should be made to:

- Learn the Evolution of Knowledge management.
- Be familiar with tools.
- Be exposed to Applications.
- Be familiar with some case studies.

UNIT I : INTRODUCTION

Introduction: An Introduction to Knowledge Management – The foundations of knowledge management- including cultural issues- technology applications organizational concepts and processes- management aspects- and decision support systems. The Evolution of Knowledge management: From Information Management to Knowledge Management – Key Challenges Facing the Evolution of Knowledge Management – Ethics for Knowledge Management.

UNIT II : CREATING THE CULTURE OF LEARNING AND KNOWLEDGE SHARING

Organization and Knowledge Management – Building the Learning Organization. Knowledge Markets: Cooperation among Distributed Technical Specialists – Tacit Knowledge and Quality Assurance.

UNIT III : KNOWLEDGE MANAGEMENT-THE TOOLS

Telecommunications and Networks in Knowledge Management – Internet Search Engines and Knowledge Management – Information Technology in Support of Knowledge Management – Knowledge Management and Vocabulary Control – Information Mapping in Information Retrieval – Information Coding in the Internet Environment – Repackaging Information.

UNIT IV : KNOWLEDGE MANAGEMENT-APPLICATION

Components of a Knowledge Strategy – Case Studies (From Library to Knowledge Center, Knowledge Management in the Health Sciences, Knowledge Management in Developing Countries).

UNIT V : FUTURE TRENDS AND CASE STUDIES

Advanced topics and case studies in knowledge management – Development of a knowledge management map/plan that is integrated with an organization's strategic and business plan – A case study on Corporate Memories for supporting various aspects in the process life -cycles of an organization.

TEXT BOOK:

- Srikantaiah, T.K., Koenig, M., "Knowledge Management for the Information Professional" Information Today, Inc., 2000.

REFERENCE:

- Nonaka, I., Takeuchi, H., "The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation", Oxford University Press, 1995.

RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Computer Science and Engineering, VI-Semester

Open Elective - CS604 (B) Project Management

Course Learning Objectives:

Understand the different activities in software project development i.e, planning, design and management.

Course content:

1. Conventional Software Management.

Evolution of software economics. Improving software economics: reducing product size, software processes, team effectiveness, automation through software environments. Principles of modern software management.

2. Software Management Process

Framework,: Life cycle phases- inception, elaboration, construction and training phase. Artifacts of the process- the artifact sets, management artifacts, engineering artifacts, pragmatics artifacts. Model based software architectures. Workflows of the process. Checkpoints of the process.

3. Software Management Disciplines

Iterative process planning. Project organisations and responsibilities. Process automation. Project control And process instrumentation- core metrics, management indicators, life cycle expectations. Process discriminants.

Books

1. Software Project management, Walker Royce, Addison Wesley, 1998.
2. Project management 2/e ,Maylor.
3. Managing the Software Process, Humphrey.
4. Managing global software Projects, Ramesh, TMH,2001.

Course Outcomes:

1. Understanding the evolution and improvement of software economics according to the basic parameters and transition to the modern software management.
2. Learning the objectives, activities and evaluation criteria of the various phases of the life cycle of software management process.
3. Gaining knowledge about the various artifacts, workflows and checkpoints of the software management process and exploring the design concept using model based architecture from technical and management perspective.
4. Develop an understanding of project planning, organization, responsibilities, automation and control of the processes to achieve the desirable results.

RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Computer Science and Engineering, VI-Semester

Open Elective - CS604 (C) Rural Technology & Community Development

Unit – I: Rural Management –

Principles and Practices Introduction to Management and Theory of Management B. Planning, Organisation Structure and Design C. Motivation and Leadership D. Management Control and Managerial Decision Making

Unit – II: Human Resource Management for rural India

Nature, Scope of Human Resource Management. F. Human Resource Planning, Recruitment and Selection, Training and Development, Performance Appraisal G. Welfare programme and Fringe benefits, Wage and Salary Administration H. Morale and Productivity, Industrial Relations and Industrial Disputes

Unit-III Management of Rural Financing:

Rural Credit System, Role of Rural Credit in Rural Development. Evolution and Growth of Rural Credit System in India. B: Agricultural Credit, Agricultural Credit Review Committee, Report of different Committees and Commissions, Problems and Prospects. C: Rural Credit to Non-farm Sector, Credit for small and marginal entrepreneurs. D: Role of Government Institutions towards facilitating Rural Credit. Role of Non- Government/ Semi Government / Quasi- Government Institutions. Growth and Present trend of Rural Financing towards Small scale and Cottage Industries.

Unit – IV: Research Methodology:

Concept of Social Research, Traditional Research, Action Research and Participatory Research B: Qualitative Data Construction and Methods of Data Collection C: Techniques of Interview D: Qualitative methods: Sociometry, Case Studies, observation, coding and content analysis

Unit – V: Research Methodology

Collection, Tabulation and Presentation of data B: Measures of Central Tendency, Dispersion, Moments, Skewness and Kurtosis, Correlation and Regression: Sampling Theory and Test of Significance

Department of Computer Science and Engineering
Subject Notes
CS-604 (B) Project Management
Unit -1

Topics to be covered

Evolution of software economics, improving software economics: reducing product size, software processes, team effectiveness, automation through software environments. Principles of modern software management.

1. Evolution of software economics

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

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

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

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

Several parametric models have been developed to estimate software costs; all of them can be generally abstracted into this form. One important aspect of software economics (as represented within today's software cost models) is that the relationship between effort and size exhibits a diseconomy of scale. The diseconomy of scale of software development is a result of the process exponent being greater than 1.0. Contrary to most manufacturing processes, the more software you build, the more expensive it is per unit item.

Target objective: improved ROI

Environment/Tools: Off-the-shelf, integrated
Size: 70% component-based 30% custom
Process: Managed/measured

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

The old way and the new: The principles of conventional software Engineering, principles of modern software management, transitioning to an iterative process.

2. 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.

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
Environment Automation technologies and tools	Integrated tools (visual modeling, compiler, editor, debugger, change management, etc.) Open systems Hardware platform performance Automation of coding, documents, testing, analyses
Quality Performance, reliability, accuracy	Hardware platform performance Demonstration-based assessment Statistical quality control

3. 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 here as the general term for reducing the "source" language size necessary 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). This size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic, and fourth-generation languages), automatic code generators (CASE tools, visual modeling tools, GUI builders), reuse of commercial components (operating systems, windowing environments, database management systems, middleware, networks), and object-oriented technologies (Unified Modeling Language, visual modeling tools, architecture

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

LANGUAGES

Universal function points (UFPs) 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.

Language Expressiveness

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

OBJECT-ORIENTED METHODS AND VISUAL MODELING

There has been a widespread movement in the 1990s toward object-oriented technology. I spend very little time on this topic because object-oriented technology is not germane to most of the software management topics discussed here, and books on object-oriented technology abound. Some studies have concluded that 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. These are interesting examples of the interrelationships among the dimensions of improving software economics.

1. An object-oriented model of the problem and its solution encourages a common vocabulary between the end users of a system and its developers, thus creating a shared understanding of the problem being solved.
2. The use of continuous integration creates opportunities to recognize risk early and make incremental corrections without destabilizing the entire development effort.
3. An object-oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire architecture.

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

1. A ruthless focus on the development of a system that provides a well understood collection of essential minimal characteristics.
2. The existence of a culture that is centred on results, encourages communication, and yet is not afraid to fail.
3. The effective use of object-oriented modelling.
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. Software design methods have always dealt implicitly with reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality. Try to treat reuse as a mundane part of achieving a return on investment.

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

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

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

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



COMMERCIAL COMPONENTS

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

Advantages and Disadvantages of Commercial Components and Custom Software

APPROACH	ADVANTAGE	DISADVANTAGE
Commercial Components	Predictable license costs Broadly used mature technology Dedicated support	Frequent upgrades Up-front license fees Recurring maintenance fees Dependency on vendor

	organization Hardware/Software independence Rich in functionality	Run time efficiency sacrifices Functionality constraints Integration not always trivial No control over upgrades and maintenance Unnecessary features that consumes extra resources Often inadequate reliability and stability Multiple vendor incompatibilities
Custom development	Complete change freedom Smaller, often simpler implementations Often better performance Control of development and enhancement	Expensive, unpredictable development Unpredictable availability date Undefined maintenance model Often immature and fragile Drain on expert resources

4. Improving Software Process

Process is an overloaded term. For software-oriented organizations, there are many processes and sub processes. 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.

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 improvement

5. Improving Team Effectiveness

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

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

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

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

- **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.
- **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

6. 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 modelling 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. Integrated toolsets play an increasingly important role in incremental/iterative development by allowing the designers to traverse quickly among development artifacts and keep them up-to-date.

Round-trip engineering is a term used to describe the key capability of environments that support iterative development. As we have moved into maintaining different information repositories for the engineering artifacts, we need automation support to ensure efficient and error-free transition of data from one artifact to another. *Forward engineering* is the automation of one engineering artifact from another, more abstract

representation. For example, compilers and linkers have provided automated transition of source code into executable code. *Reverse engineering* is the generation or modification of a more abstract representation from an existing artifact (for example, creating a visual design model from a source code representation). Economic improvements associated with tools and environments. It is common for tool vendors to make relatively accurate individual assessments of life-cycle activities to support claims about the potential economic impact of their tools. For example, it is easy to find statements such as the following from companies in a particular tool.

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

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 modelling 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

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

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

PEER INSPECTIONS: A PRAGMATIC VIEW

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

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

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

THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

1. **Make quality #1.** Quality must be quantified and mechanisms put into place to motivate its achievement
2. **High-quality software is possible.** Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people
3. **Give products to customers early.** No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it
4. **Determine the problem before writing the requirements.** When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution
5. **Evaluate design alternatives.** After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model.** Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases.** Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation throughout the life cycle.
8. **Minimize intellectual distance.** To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure

9. **Put techniques before tools.** An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer
10. **Get it right before you make it faster.** It is far easier to make a working program run faster than it is to make a fast program work. Don't worry about optimization during initial coding
11. **Inspect code.** Inspecting the detailed design and code is a much better way to find errors than testing
12. **Good management is more important than good technology.** Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success.** Highly skilled people with appropriate experience, talent, and training are key.
14. **Follow with care.** Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.
15. **Take responsibility.** When a bridge collapses we ask, "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.
16. **Understand the customer's priorities.** It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.
17. **The more they see, the more they need.** The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.
18. **Plan to throw one away.** One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first time.
19. **Design for change.** The architectures, components, and specification techniques you use must accommodate change.
20. **Design without documentation is not design.** I have often heard software engineers say, "I have finished the design. All that is left is the documentation. "
21. **Use tools, but be realistic.** Software tools make their users more efficient.
22. **Avoid tricks.** Many programmers love to create programs with tricks constructs that perform a function correctly, but in an obscure way. Show the world how smart you are by avoiding tricky code
23. **Encapsulate.** Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.
24. **Use coupling and cohesion.** Coupling and cohesion are the best ways to measure software's inherent maintainability and adaptability.
25. **Use the McCabe complexity measure.** Although there are many metrics available to report the inherent complexity of software, none is as intuitive and easy to use as Tom McCabe's.

7. Principles of Modern Software Management

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

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, 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.
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).
 1. **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.
 2. **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.
 3. **Use a *demonstration-based approach* to assess intermediate artifacts. 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.
6. **Establish a *configurable process* that is economically scalable.** No single process is suitable for all software developments.

TRANSITIONING TO AN ITERATIVE PROCESS

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

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

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

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

iterations in the life cycle establish precedents from which the product, the process, and the plans can be elaborated in **evolving levels of detail**.

- **Process flexibility.** Development of modern software is characterized by such a broad solution space and so many interrelated concerns that there is a paramount need for continuous incorporation of changes. These changes may be inherent in the problem understanding, the solution space, or the plans. Project artifacts must be supported by efficient **change management** commensurate with project needs. A **configurable process** that allows a common framework to be adapted across a range of projects is necessary to achieve a software return on investment.
- **Architecture risk resolution.** **Architecture-first** development is a crucial theme underlying a successful iterative development process. A project team develops and stabilizes architecture before developing all the components that make up the entire suite of applications components. An **architecture-first** and **component-based development approach** forces the infrastructure, common mechanisms, and control mechanisms to be elaborated early in the life cycle and drives all component make/buy decisions into the architecture process.
- **Team cohesion.** Successful teams are cohesive, and cohesive teams are successful. Successful teams and cohesive teams share common objectives and priorities. Advances in technology (such as programming languages, UML, and visual modelling) 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**.

Department of Computer Science and Engineering
Subject Notes
CS-604 (B) Project Management
Unit -2

Topics to be covered

Software Management Process

Framework: Life cycle phases- inception, elaboration, construction and transition phase. Artifacts of the process- the artifact sets, management artifacts, engineering artifacts, pragmatics artifacts. Model based software architectures. Workflows of the process. Checkpoints of the process.

Engineering and Production Stages:

To achieve economies of scale and higher return on investment, we must move toward a software manufacturing process which is determined by technological improvements in process automation and component based development.

There are two stages in the software development process:


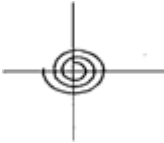
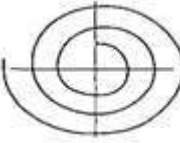
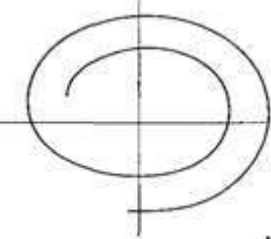
- 1) **The Engineering Stage:** Less predictable but smaller teams doing design and production activities. This stage is decomposed into two distinct Phases inception and elaboration.
- 2) **The Production Stage:** More predictable but larger teams doing construction, test, and deployment activities. This stage is also decomposed into two distinct Phases construction and transition.

Table 2.1 the Two Stages of the Life Cycle: Engineering and Production

S no.	Life Cycle Aspects	Engineering Stage Emphasis	Production Stage Emphasis
1	Risk Reduction	Schedule, Technical, Feasibility	Cost
2	Products	Architecture baseline	Product release baseline
3	Activities	Analysis, Design, Planning	Implementation, Testing
4	Assessment	Demonstration, Inspection, Analysis	Testing
5	Economics	Resolving diseconomies of scale	Exploiting Economies of scale
6	Management	Planning	Operations

These four phases of lifecycle process are loosely mapped to the conceptual framework of the spiral model is as shown in the following table:

Table 2.2 Four phases of lifecycle

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

- In the above figure the size of the spiral corresponds to the inactivity of the project with respect to the breadth and depth of the artifacts that have been developed.
- This inertia manifests itself in maintaining artifact consistency, regression testing, documentation, quality analyses, and configuration control.
- Increased inertia may have little, or at least very straightforward, impact on changing any given discrete component or activity.
- However, the reaction time for accommodating major architectural changes, major requirements changes, major planning shifts, or major organizational perturbations clearly increases in subsequent phases.

Inception Phase:

The main goal of this phase is to achieve agreement among stakeholders on the life-cycle objectives for the project.

Primary Objectives:

- 1) Establishing the project's scope and boundary conditions
- 2) Distinguishing the critical use cases of the system and the primary scenarios of operation
- 3) Demonstrating at least one candidate architecture against some of the primary scenarios
- 4) Estimating cost and schedule for the entire project
- 5) Estimating potential risks

Essential Activities:

- 1) Formulating the scope of the project
- 2) Synthesizing the architecture
- 3) Planning and preparing a business case

Elaboration Phase:

- It is the most critical phase among the four phases.
- Depending upon the scope, size, risk, and freshness of the project, an executable architecture prototype is built in one or more iterations.
- At most of the time the process may accommodate changes, the elaboration phase activities must ensure that the architecture, requirements, and plans are stable. And also the cost and schedule for the completion of the development can be predicted within an acceptable range.

Primary Objectives

- 1) Base lining the architecture as rapidly as practical
- 2) Base lining the vision
- 3) Base lining a high-reliability plan for the construction phase
- 4) Demonstrating that the baseline architecture will support the vision at a reasonable cost in a reasonable time.

Essential Activities

- 1) Elaborating the vision
- 2) Elaborating the process and infrastructure
- 3) Elaborating the architecture and selecting components

Construction Phase

During this phase all the remaining components and application features are integrated into the application and all features are thoroughly tested. Newly developed software is integrated where ever required. If it is a big project then parallel construction increments are generated.

Primary Objectives

- 1) Minimizing development costs
- 2) Achieving adequate quality as rapidly as practical
- 3) Achieving useful version (alpha, beta, and other releases) as rapidly as practical

Essential Activities

- 1) Resource management, control, and process optimization
- 2) Complete component development and testing evaluation criteria
- 3) Assessment of product release criteria of the vision

Transition Phase

Whenever a project is grown-up completely and to be deployed in the end-user domain this phase is called transition phase. It includes 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

Primary Objectives

- 1) Achieving user self-supportability
- 2) Achieving stakeholder concurrence
- 3) Achieving final product baseline as rapidly and cost-effectively as practical

Essential Activities

- 1) Synchronization and integration of concurrent construction increments into consistent deployment baselines
- 2) Deployment-specific engineering
- 3) Assessment of deployment baselines against the complete vision and acceptance criteria in the requirement set.

Artifacts of the Process:

- Conventional s/w projects focused on the sequential development of s/w artifacts:
- Build the requirements
- Construct a design model traceable to the requirements
- Compile and test the implementation for deployment.
- This process can work for small-scale, purely custom developments in which the design representation, implementation representation and deployment representation are closely aligned.
- This approach is doesn't work for most of today's s/w systems why because of having complexity and are composed of numerous components some are custom, some reused, some commercial products.

The Artifacts Sets

In order to manage the development of a complete software system, we need to gather distinct collections of information and is organized into artifact sets.

- Set represents a complete aspect of the system where as artifact represents interrelated information that is developed and reviewed as a single entity.
- The artifacts of the process are organized into five sets:
1) Management 2) Requirements 3) Design 4) Implementation 5) Deployment

Here the management artifacts capture the information that is necessary to synchronize stakeholder expectations. Whereas the remaining four artifacts are captured rigorous notations that support automated analysis and browsing

Table 2.3 The Artifacts Sets

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 baseline 2. Associated Compile-time files 3. Component executables	1. Integrated product executable baseline 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		1. Release descriptions 2. Status assessments 3. Software change order database 4. Deployment documents and Environment	

The Management Set

It captures the artifacts associated with process planning and execution. These artifacts use ad hoc notation including text, graphics, or whatever representation is required to capture the “contracts” among,

- **project personnel:** project manager, architects, developers, testers, marketers, administrators
- **stakeholders:** Funding authority, user, s/w project manager, organization manager, regulatory agency & between project personnel and stakeholders

Management artifacts are evaluated, assessed, and measured through a combination of

- 1) Relevant stakeholder review.
- 2) Analysis of changes between the current version of the artifact and previous versions.
- 3) Major milestone demonstrations of the balance among all artifacts.

The Engineering Sets:

1) Requirement Set:

- The requirement set is the primary engineering context for evaluating the other three engineering artifact sets and is the basis for test cases.
- **Requirement artifacts are evaluated, assessed, and measured through a combination of**

- 1) Analysis of consistency with the release specifications of the mgmt set.
- 2) Analysis of consistency between the vision and the requirement models.
- 3) Mapping against the design, implementation, and deployment sets to evaluate the consistency and completeness and the semantic balance between information in the different sets.
- 4) Analysis of changes between the current version of the artifacts and previous versions.
- 5) Subjective review of other dimensions of quality.

2) Design Set:

- UML notations are used to engineer the design models for the solution.
- It contains various levels of abstraction and enough structural and behavioral information to determine a bill of materials.
- Design model information can be clearly and, in many cases, automatically translated into a subset of the implementation and deployment set artifacts.

The design set is evaluated, assessed, and measured through a combination of

- 1) Analysis of the internal consistency and quality of the design model.
- 2) Analysis of consistency with the requirements models.
- 3) Translation into implementation and deployment sets and notations to evaluate the Consistency and completeness and semantic balance between information in the sets.
- 4) Analysis of changes between the current version of the design model and previous versions.
- 5) Subjective review of other dimensions of quality.

3) Implementation set:

The implementation set include source code that represents the tangible implementations of components and any executables necessary for stand-alone testing of components.

- Executables are the primitive parts that are needed to construct the end product, including custom components, APIs of commercial components.
- Implementation set artifacts can also be translated into a subset of the deployment set.

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

- 1) Analysis of consistency with design models
- 2) Translation into deployment set notations to evaluate consistency and completeness among artifact sets.
- 3) Execution of stand-alone component test cases that automatically compare expected results with actual results.
- 4) Analysis of changes b/w the current version of the implementation set and previous versions.
- 5) Subjective review of other dimensions of quality.

4) Deployment Set:

It 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:

- 1) 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.
- 2) Testing the partitioning, replication, and allocation strategies in mapping components of the implementation set to physical resources of the deployment system.
- 3) Testing against the defined usage scenarios in the user manual such as installation, user-oriented dynamic reconfiguration, mainstream usage, and anomaly management.
- 4) Analysis of changes b/w the current version of the deployment set and previous versions.
- 5) Subjective review of other dimensions of quality.

Each artifact set uses different notations to capture the relevant artifact.

- 1) **Management set notations** (ad hoc text, graphics, use case notation) capture the plans, process, objectives, and acceptance criteria.
- 2) **Requirement notation** (structured text and UML models) capture the engineering context and the operational concept.
- 3) **Implementation notations** (software languages) capture the building blocks of the solution in human-readable formats.
- 4) **Deployment notations** (executables and data files) capture the solution in machine-readable formats.

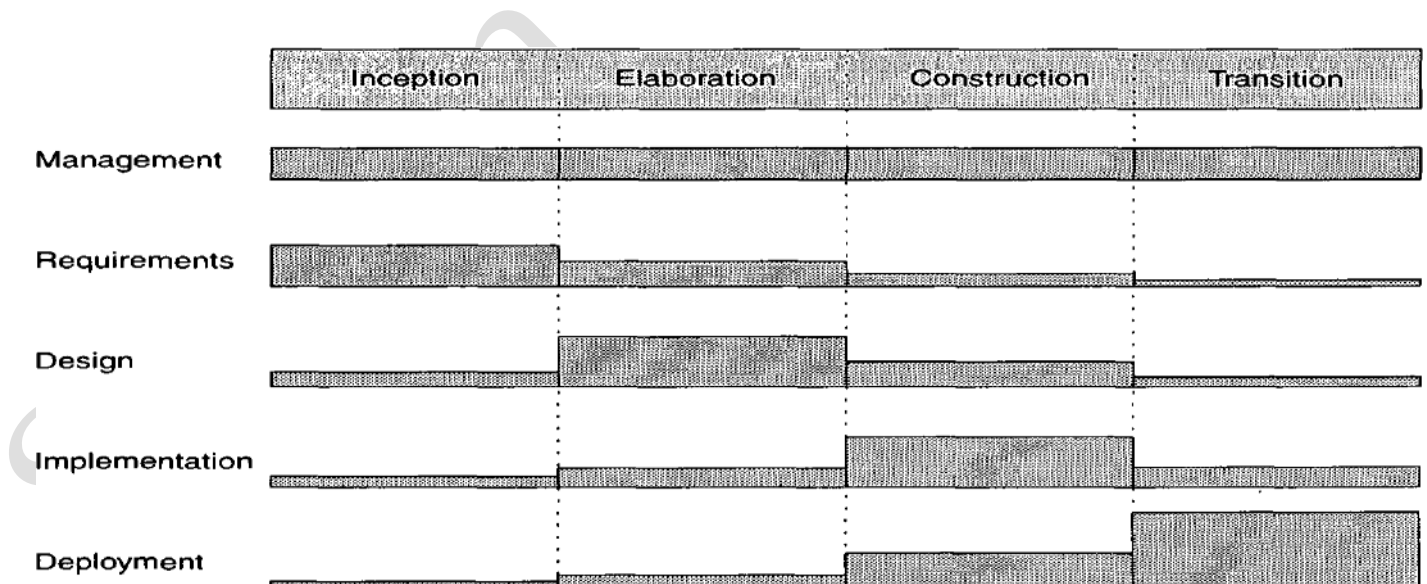


Figure 2.1 Different notations to capture the relevant artifact

Implementation Set verses Deployment Set

- A. The structure of the information delivered to the user (testing people) is very different from the structure of the source code implementation.
- B. Engineering decisions that have impact on the quality of the deployment set but are relatively incomprehensible in the design and implementation sets include:

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

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.

Software Development Plan

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

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.

Pragmatic Artifacts

- People want to review information but don't understand the language of the artifact. Many interested reviewers of a particular artifact will resist having to learn the engineering language in which the artifact is written. It is not uncommon to find people (such as veteran software managers, veteran quality assurance specialists, or an auditing authority from a regulatory agency) who react as follows: "I'm not going to learn UML, but I want to review the design of this software, so give me a separate description such as some flowcharts and text that I can understand."
- People want to review the information but don't have access to the tools. It is not very common for the development organization to be fully tooled; it is extremely rare that the/other stakeholders have any Capability to review the engineering artifacts on-line. Consequently, organizations are forced to exchange paper documents. Standardized formats (such as UML, spreadsheets, Visual Basic, C++, and Ada 95), visualization tools, and the Web are rapidly making it economically feasible for all stakeholders to exchange information electronically
- Human-readable engineering artifacts should use rigorous notations that are complete, consistent, and used in a self-documenting manner. Properly spelled English words should be used for all identifiers and descriptions. Acronyms and abbreviations should be used only where they are well accepted jargon in the context of the component's usage. Readability should be emphasized and the use of proper English words should be required in all engineering artifacts. This practice enables understandable representations, browse able formats (paperless review), more-rigorous notations, and reduced error rates.
- Useful documentation is self-defining: It is documentation that gets used.
- Paper is tangible; electronic artifacts are too easy to change. On-line and Web-based artifacts can be changed easily and are viewed with more skepticism because of their inherent volatility.

Model-Based Software Architectures

- Software architecture is the central design problem of a complex software system in the same way architecture is the software system design.
- The ultimate goal of the engineering stage is to converge on a stable architecture baseline.
- Architecture is not a paper document. It is a collection of information across all the engineering sets.
- Architectures are described by extracting the essential information from the design models.
- A model is a relatively independent abstraction of a system.
- A view is a subset of a model that abstracts a specific, relevant perspective.

Architecture: Management Perspective

The most critical and technical product of a software project is its architecture

- If a software development team is to be successful, the inter project communications, as captured in software architecture, must be accurate and precise.

From the management point of view, three different aspects of architecture

1. An architecture (the intangible design concept) is the design of software system it includes all engineering necessary to specify a complete bill of materials. Significant make or buy decisions are resolved, and all custom components are elaborated so that individual component costs and construction/assembly costs can be determined with confidence.
2. An architecture baseline (the tangible artifacts) is a slice of information across the engineering artifact sets sufficient to satisfy all stakeholders that the vision (function and quality) can be achieved within the parameters of the business case (cost, profit, time, technology, and people).
3. An architectural description is an organized subset of information extracted from the design set model's. It explains how the intangible concept is realized in the tangible artifacts.

The number of views and level of detail in each view can vary widely. For example the architecture of the software architecture of a small development tool.

There is a close relationship between software architecture and the modern software development Process because of the following reasons:

1. A stable software architecture is nothing but a project milestone where critical make/buy decisions should have been resolved. The life-cycle represents a transition from the engineering stage of a project to the production stage.
2. Architecture representation provide a basis for balancing the trade-offs between the problem space (requirements and constraints) and the solution space (the operational product).
3. The architecture and process encapsulate many of the important communications among individuals, teams, organizations, and stakeholders.
4. Poor architecture and immature process are often given as reasons for project failure.
5. In order to proper planning, a mature process, understanding the primary requirements and demonstrable architecture are important fundamentals.
6. Architecture development and process definition are the intellectual steps that map the problem to a solution without violating the constraints; they require human innovation and cannot be automated.

Architecture: Technical Perspective

Software architecture includes the structure of software systems, their behavior, and the patterns that guide these elements, their collaborations, and their composition. An architecture framework is defined in terms of views is the abstraction of the UML models in the design set. Where as architecture view is an abstraction of the design model, include full breadth and Depth of information.

Most real-world systems require four types of views:

- 1) Design: describes architecturally significant structures and functions of the design model.
- 2) Process: describes concurrency and control thread relationships among the design, component, and deployment views.
- 3) Component: describes the structure of the implementation set.
- 4) Deployment: describes the structure of the deployment set.

Workflows of the Process:

The term **workflow** means a thread of cohesive and mostly sequential activities. In most of the cases a process is a sequence of activities why because of easy to understand, represent, plan and conduct. But the simplistic activity sequences are not realistic why because it includes many teams, making progress on many artifacts that must be synchronized, cross-checked, homogenized, merged and integrated. In order to manage complex software's the workflow of the software process is to be changed that is distributed. Modern software process avoids the life-cycle phases like inception, elaboration, construction and transition. It tells only the state of the project rather than a sequence of activities in each phase. The activities of the process are organized in to seven major workflows:

- 1) Management
- 2) Environment
- 3) Requirements
- 4) Design
- 5) Implementation
- 6) Assessment
- 7) Deployment

Management workflow: controlling the process and ensuring win conditions for all stakeholders.

Environment workflow: automating the process and evolving the maintenance environment.

Requirements workflow: analyzing the problem space and evolving the requirements artifacts.

Design workflow: modeling the solution and evolving the architecture and design artifacts.

Implementation workflow: programming the components and evolving the implementation and deployment artifacts.

Assessment workflow: assessing the trends in process and product quality.

Deployment workflow: transitioning the end products to the user.

Table 2.4 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 Component Transition: Maintain components
Assessment	Release specifications Release Descriptions User manuals Deployment set	Inception: Assess plans, vision, prototypes Elaboration: Assess architecture Construction: Assess internal releases Transition: Assess producer releases
Deployment	Deployment Set	Inception: Analyze user community Elaboration: Design user manual Construction: Prepare transition materials Transition: Transition product to user

Checkpoints of the Process:

It is important to place visible milestones in the life cycle in order to discuss the progress of the project by the Stakeholders and also to achieve,

- 1) Synchronize stakeholder expectations and achieve agreement among the requirements, the design, and the plan perspectives.
- 2) Synchronize related artifacts into a consistent and balanced state.
- 3) Identify the important risks, issues, and out-of-tolerance conditions.
- 4) Perform a global review for the whole life cycle, not just the current situation of an individual perspective or intermediate product.

Three sequences of project checkpoints are used to synchronize stakeholder expectations throughout the Lifecycle:

- 1) **Major milestones**
- 2) **Minor milestones**
- 3) **Status assessments**

1) Major Milestones:

The most important major milestone is usually the event that transitions the project from the elaboration phase into the construction phase. These are the system wide events are held at the end of each development phase. They provide visibility to system wide issues. Major milestones at the end of each phase use formal, stakeholder-approved evaluation criteria and release descriptions. 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 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, tradeoff 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: regulatory agencies, independent verification and validation contractors, venture capital investors, subcontractors, associate contractors, and sales and marketing teams.

2) Minor Milestones: The format and content of minor milestones are highly dependent on the project and the organizational culture. These are the iteration-focused events are conducted to review the content of an iteration in detail and to authorize continued work. Minor milestones capture two artifacts: a release specification and a release description. Minor milestones use informal, development-team-controlled versions of these artifacts.

- The number of iteration-specific, informal milestones needed depends on the content and length of the iteration.
- Iterations which have one-month to six-month duration have only two milestones are needed: the iteration readiness review and iteration assessment review. For longer iterations some other intermediate review points are added.
- All iterations are not created equal . An iteration take different forms and priorities, depending on where the project is in the life cycle.
- Early iterations focus on analysis and design. Later iterations focus on completeness, consistency, usability and change management.
- **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 to review iteration results, test results, to determine amount of rework to be done, to review impact of the iteration results on the plan for subsequent iterations.

3) Status Assessments: Periodic status assessments are important for focusing continuous attention on the evolving health of the project and its dynamic priorities. These are periodic events provide management with frequent and regular insight into the progress being made. These are management reviews conducted at regular intervals (monthly, quarterly) to address progress and quality of project and maintain open communication among all stakeholders. The main objective of this assessment is to synchronize all Stakeholders expectations and also serve as project snapshots. Also provide,

- 1) A mechanism for openly addressing, communicating, and resolving management issues, technical issues, and project risks.
- 2) A mechanism for broadcast process, progress, quality trends, practices, and experience information to and from all stakeholders in an open forum.
- 3) Objective data derived directly from on-going activities and evolving product configurations.

Department of Computer Science and Engineering
Subject Notes
CS-604 (B) Project Management
Unit -3

Topics to be covered

Iterative process planning. Project organizations and responsibilities. Process automation. Project control and process instrumentation- core metrics, management indicators, life cycle expectations. Process discriminate

1. Iterative Process Planning

1.1 Work Breakdown Structures

The development of a work breakdown structure is dependent on the project management style, organizational culture, customer preference, financial constraints and several other hard-to-define 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:

1. A delineation of all significant work.
2. A clear task decomposition for assignment of responsibilities.
3. A framework for scheduling, budgeting, and expenditure tracking.

Two simple planning guidelines should be considered when a project plan is being initiated or assessed.

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

Table 3.1 The First Guideline Prescribes a Default Allocation of Costs Among The First-Level WBS Elements

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

Table 3.2 The Second Guideline Prescribes the Allocation of Effort and Schedule Across the Life-Cycle Phases

1.2 The Cost and Schedule Estimating Process: -

Forward-looking: 1. The software project manager 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, also partitions the schedule into major milestone dates and partitions the effort into a staffing profile.

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.

Backward-looking:

1. The lowest level WBS elements are elaborated into detailed tasks, for which budgets and schedules are estimated by the responsible WBS element manager.
2. Estimates are combined and integrated into higher level budgets and milestones.
3. Comparisons are made with the top-down budgets and schedule milestones. Gross differences are assessed and adjustments are made in order to converge on agreement between the top-down and the bottom-up estimates.

1.3. The Iteration Planning Process

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

Table 3.3 The Iteration Planning Process

Engineering stage planning emphasis:

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

Production stage planning emphasis:

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

2. 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 & probabilities.
- **Project teams** are motivated by the cost, Schedule & quality of specific deliverables

2.1 Line-Of-Business Organizations: The main features of default organization are as follows:

- Responsibility for process definition and maintenance is specific to a cohesive line of business, where process commonality makes sense. For example, the process for developing avionics software is different from the process used to develop office applications.
- Responsibility for process automation is an organizational role and is equal in importance to the process definition role. Projects achieve process commonality primarily through the environment support of common tools.
- Organizational roles may be fulfilled by a single individual or several different teams, depending on the scale of the organization. A 20-person software product company may require only a single

person to fulfill all the roles, while a 10,000-person telecommunications company may require hundreds of people to achieve an effective software organization.

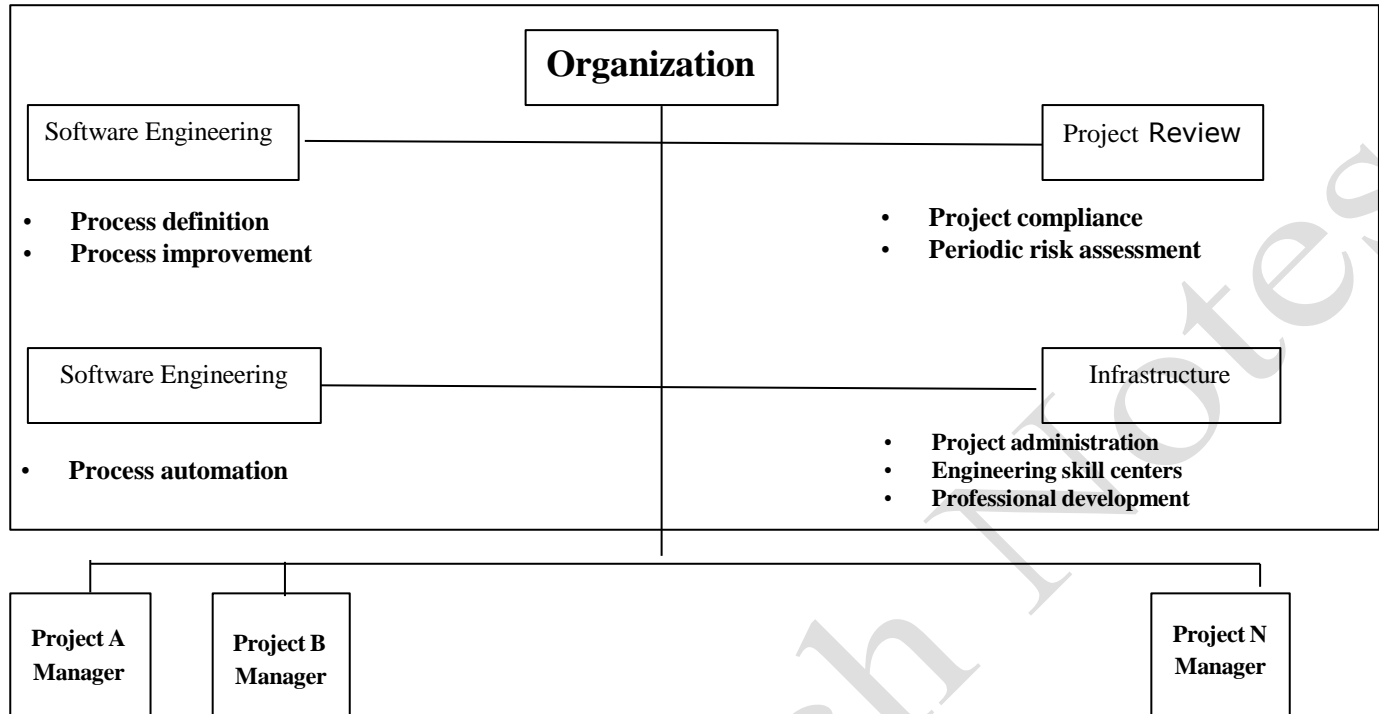


Fig 3.1 Default roles in a software Line-of-Business Organization

Software Engineering Process Authority (SEPA)

The SEPA facilitates the exchange of information & process guidance both to & from project practitioners. This role is accountable to General Manager for maintaining a current assessment of the Organization's process maturity & s 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 un software policies, practices & standards. A software Project Manager is responsible for meeting the requirements of a contract or some other project compliance standard.

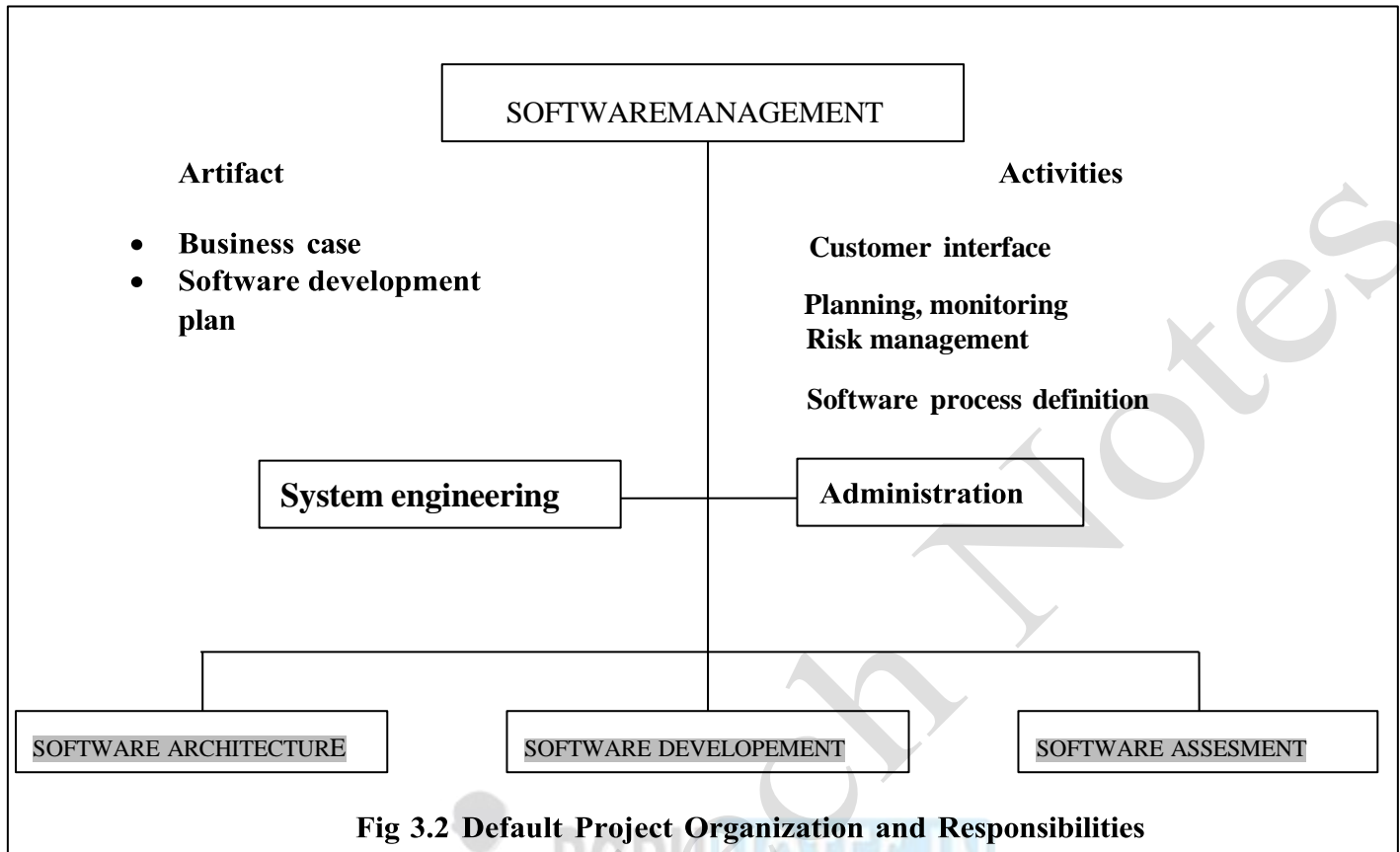
Software Engineering Environment Authority (SEEA)

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 SEE A role is necessary to achieve a significant ROI for common process.

2.2 Infrastructure

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

Project Organizations



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

EVOLUTIONS OF ORGANIZATIONS

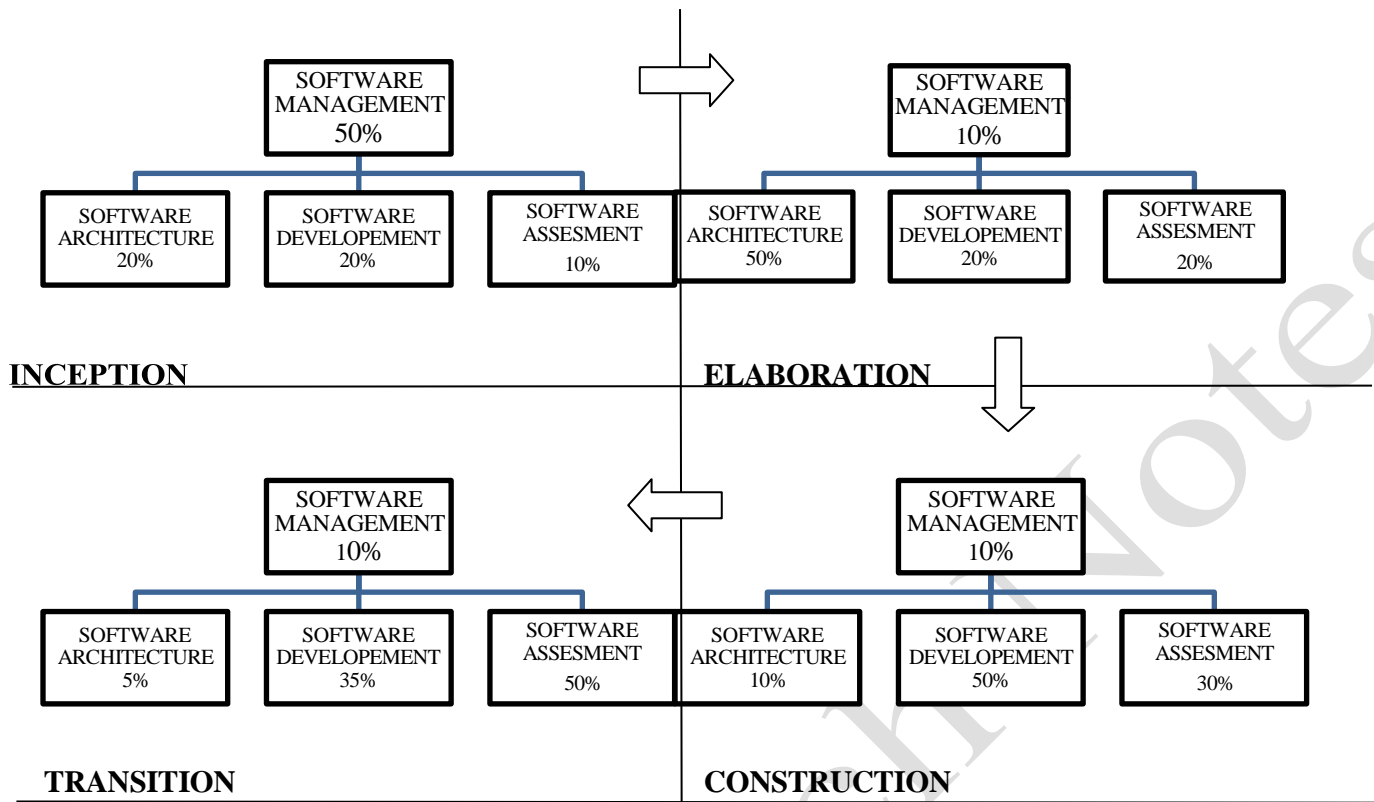


Fig 3.3 EVOLUTIONS OF ORGANIZATIONS

3. The Process Automation

Introductory Remarks:

The environment must be the first-class artifact of the process. Process automation & change management is crucial to an iterative process. If the change is expensive then the development organization will resist. Round-trip engineering & integrated environments promote change freedom & effective evolution often technical artifacts. Metric automation is crucial to effective project control. External stakeholders need access to environment resources to improve interaction with the development team & add value to the process. The three levels of process which requires a certain degree of process automation for the corresponding process to be carried out efficiently.

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

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

3. Micro process (Iteration): The automation support for generating artifacts is generally called a tool.

Automation Building blocks:

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

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

Table 3.4 Typical Automation and Tool Components that Supports Process Workflows

3.1 The Project Environment: The project environment artifacts evolve through three discrete states.

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

The Prototype Environment includes an architecture test bed for prototyping project architecture to evaluate trade-offs during inception & elaboration phase of the life cycle. The Development environment should include a full use of development tools needed to support various Process workflows & round-trip engineering to the maximum extent possible. The Maintenance Environment should typically coincide with the mature version of the development. There are four important environment disciplines that are critical to management context & the success of a modern iterative development process.

Round-Trip Engineering

Change Management

Software Change Orders (SCO)

Configuration baseline Configuration Control Board

Infrastructure

Organization Policy

Organization Environment

Stakeholder Environment.

Organization Policy

Organization Environment

Stakeholder Environment.

Round Trip Environment

Tools must be integrated to maintain consistency & traceability. Round-Trip engineering is the term used to describe this key requirement for environment that support iterative development. As the software industry moves into maintaining different information sets for the engineering artifacts, more automation support is needed to ensure efficient & error free transition of data from one artifacts to another. Round-trip engineering is the environment support necessary to maintain Consistency among the engineering artifacts.

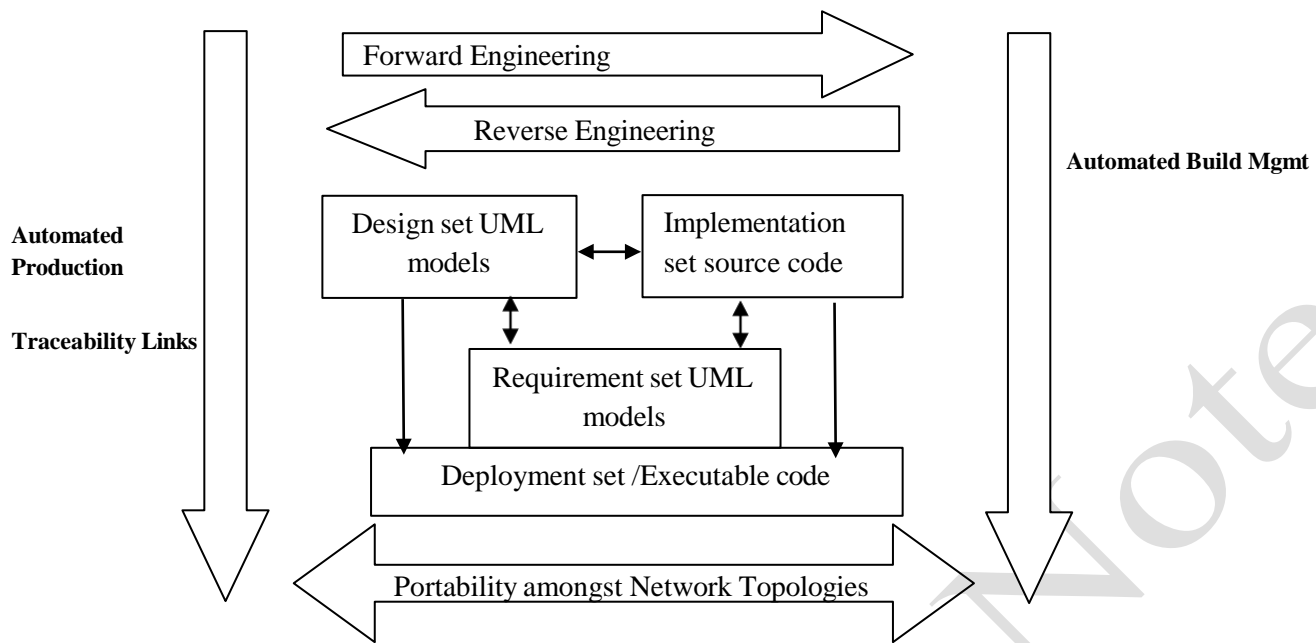


Fig 3.4 Round Trip Engineering

Change Management

Change management must be automated & enforced to manage multiple iterations & to enable change freedom. Change is the fundamental primitive of iterative development.

I. Software Change Orders

The atomic unit of software work that is authorized to create, modify or obsolesce components within configuration baseline is called a software change orders (SCO).

The basic fields of the SCO are Title, description, metrics, resolution, assessment & disposition.

II. Configuration Baseline

A configuration baseline is a named collection of software components & supporting documentation that is subjected to change management & is upgraded, maintained, tested, statuses. There are generally two classes of baselines

External Product Release

Internal testing Release

Three levels of baseline releases are required for most Systems

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

Major release represents a new generation of the product or project. A **minor** release represents the same basic product but with enhanced features, performance or quality. **Major & Minor** releases are intended to be external product releases that are persistent & supported for a period of time. An **interim** release corresponds to a developmental configuration that is intended to be transient. Once software is placed in a controlled baseline all changes are tracked such that a distinction must be made for the cause of the change. Change categories are:-

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

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

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

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

Type 3: A change that is associated by the update to the environment

Type 4: Changes that are not accommodated by the other categories.

III Configuration Control Board (CCB)

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

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

nfrastructure

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

I Organization Policy:

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

- a. Major milestones
- b. Intermediate Artifacts
- c. Engineering repository
- d. Metrics
- e. Roles & Responsibilities

Infrastructure

II Organization Environment

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

Stakeholder Environment

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

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

These stakeholder representatives also need to access to development resources so that they can contribute value to overall effort. These stakeholders will be access through on-line. An on-line environment accessible by the external stakeholders allows them to participate in the process a follow Accept & use executable increments for the hands-on evaluation. Use the same on-line tools, data & reports that the development organization uses to manage & monitor the project. Avoid excessive travel, paper interchange delays, format translations, paper shipping costs & other overhead cost.

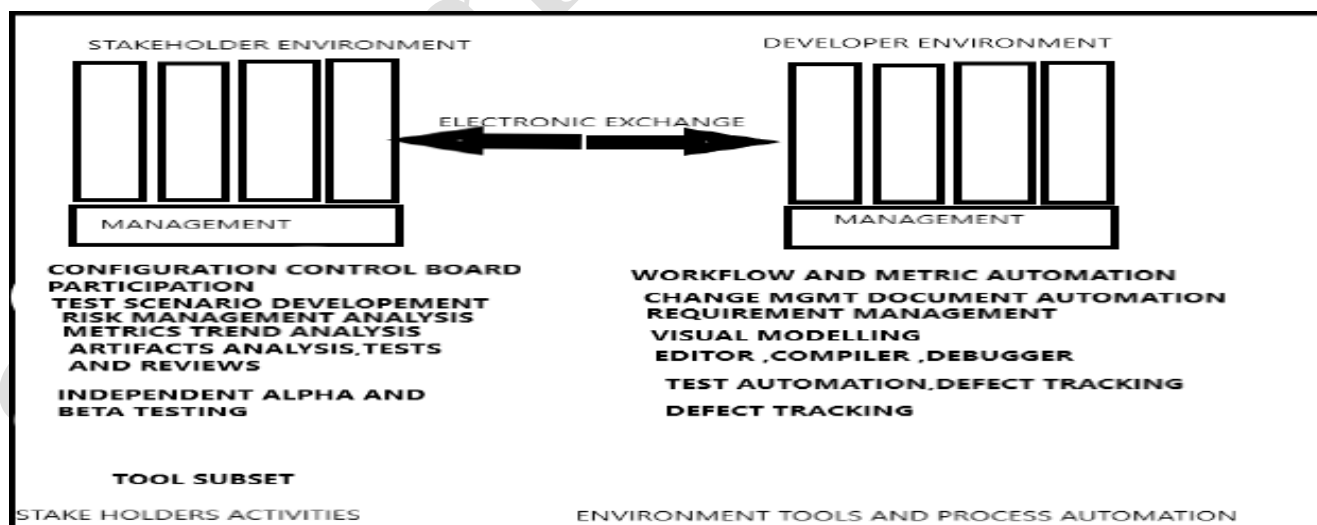


Fig 3.5 Extending environments into stakeholders domain

4. Project control and process instrumentation

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:

1. Software metrics are needed for calculating the cost and schedule of a software product with great accuracy.
2. Software metrics are required for making an accurate estimation of the progress.
3. 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.

5. SEVEN CORE METRICS OF SOFTWARE PROJECT

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. 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.

Management Indicators

Work and Progress
Budgeted cost and expenditures
Staffing and team dynamics

Quality Indicators

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

The seven-core metrics can be used in numerous ways to help manage projects and organizations. In an iterative development project or an organization structured around a software line of business, the historical values of previous iterations and projects provide precedent data for planning subsequent iterations and projects. Consequently, once metrics collection is ingrained, a project or organization can improve its ability to predict the cost, schedule, or quality performance of future work activities.

The seven-core metrics are based on common sense and field experience with both successful and unsuccessful metrics programs. Their attributes include the following:

- They are simple, objective, easy to collect, easy to interpret, and hard to misinterpret.
- Collection can be automated and nonintrusive.

- They provide for consistent assessments throughout the life cycle and are derived from the evolving product baselines rather than from a subjective assessment.
- They are useful to both management and engineering personnel for communicating progress and quality in a consistent format.
- Their fidelity improves across the life cycle.

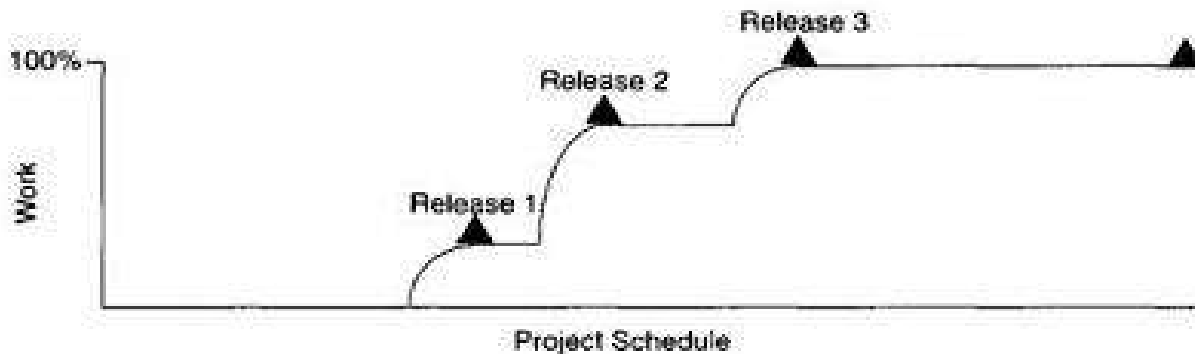


FIG 3.6 Expected Progress for typical project with three major releases

6. MANAGEMENT INDICATORS:

Work and Progress

This metric measures 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 over time) 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 met.

Software Management team: - milestones completed.

Budgeted Cost and Expenditures: - releases, by term, by components, by subsystems, etc

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 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 spending profile.

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

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

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

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

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

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

QUALITY INDICATORS:

The four quality indicators are based primarily on the measurement of software change across evolving baselines of engineering data (such as design models and source code).

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

Breakage and modularity

This metric measures the average breakage per change over time. Breakage is defined as the average extent of change, which is the amount of software baseline that needs rework and measured in source lines of code, function points, components, subsystems, files or other units. Modularity is the average breakage trend over time. This metric can be collected by revoke SLOC per change, by change type, by release, by components and by subsystems.

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 measure defect rate over time. MTBF (Mean Time between Failures) is the average usage time between software faults. It is computed by dividing the test hours by the error 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 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.

7. Life Cycle Expectations

There is no mathematical or formal derivation for using seven core metrics properly. However, there were specific reasons for selecting them: The quality indicators are derived from the evolving product rather than

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

Metric	Elaboration	Construction	Transition	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

Table 3.5 The default pattern of life cycle evolution

8. Process Discriminate

In tailoring the management process to a specific domain or projects, there are two discriminating factors: -1. Technical complexity 2. Management complexity. The formality of reviews, the quality of artifacts, the priorities of concerns and numerous other process instantiation parameters are governed by a point project occupies in these two dimensions. A well-defined software process is critical for success in software projects. Software process tailoring refers to the activity of tuning a standardized process to meet the needs of a specific project. Tailoring is necessary because each project is unique; not every process, tool, technique, input, or output identified is required on every project. Tailoring should address the competing constraints of scope, schedule, cost, resources, quality, and risk. A process framework must be configured to the specific characteristic of the project. The Process discriminants are organized around six process parameters –scale, stake holder cohesion, process flexibility, process maturity, architectural risk ad domain experience.

1. Scale: The scale of the project is the team size, which drives the process configuration more than any other factor. There are many ways to measure scale, including number of sources lines of code, number of function

points, number of use cases and number of dollars. The primary measure of scale is the size of the team. Five people are an optimal size for an engineering team.

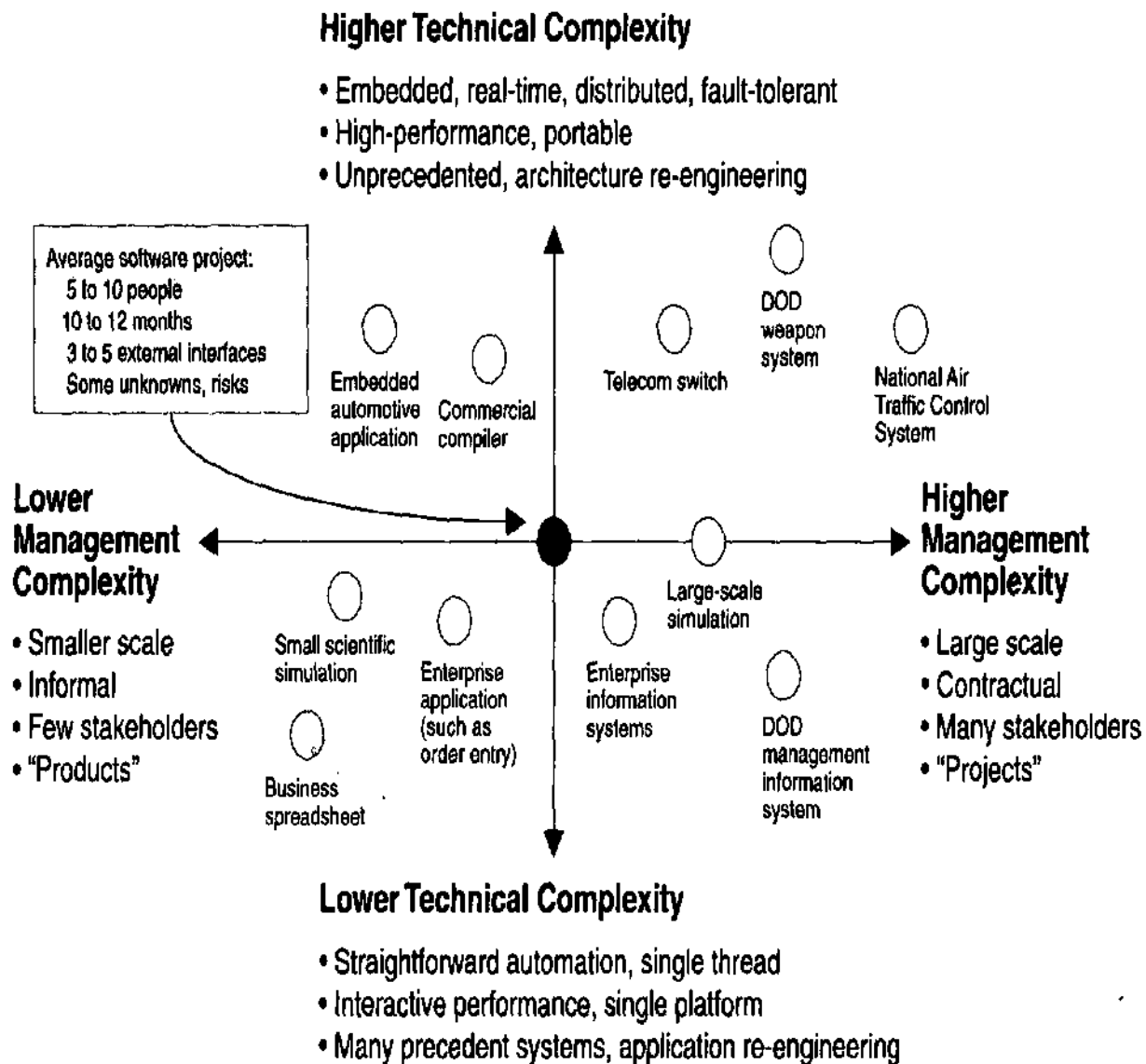


Fig 3.7 Priorities for Tailoring a Process Framework

2. Stakeholder Cohesion or Contention: The degree of cooperation and coordination among stakeholders (buyers, developers, users, subcontractors and maintainers) significantly drives the specifics of how a process is defined. This process parameter ranges from cohesive to adversarial. Cohesive teams have common goals, complementary skills and close communications. Adversarial teams have conflicting goals, competing and incomplete skills, and less-than-open communication.

3. Process Flexibility or Rigor: The implementation of the project's process depends on the degree of rigor, formality and change freedom evolved from projects contract (vision document, business case and development plan). For very loose contracts such as building a commercial product within a business unit of a software company, management complexity is minimal. For a very rigorous contract, it could take many months to authorize a change in a release schedule.

4. Process Maturity: The process maturity level of the development organization is the key driver of management complexity. Managing a mature process is very simpler than managing an immature process. Organization with a mature process have a high level of precedent experience in developing software and a high level of existing process collateral that enables predictable planning and execution of the process. This sort of collateral includes well-defined methods, process automation tools, and trained personnel, planning metrics, artifact templates and workflow templates.

5. Architectural Risk: The degree of technical feasibility is an important dimension of defining a specific projects process. There are many sources of architecture risk. They are (1) system performance which includes resource utilization, response time, throughout and accuracy, (2) robustness to change which includes addition of new features & incorporation of new technology and (3) system reliability which includes predictable behaviour and fault tolerance.

6. Domain Experience: The development organization's domain experience governs its ability to converge on an acceptable architecture in a minimum no of iterations.

A process framework is not a project specific process implementation with a well defined recipe of success. Judgement must be injected and the methods, techniques, culture, formality, organization must be tailored to the specific domain to achieve a process implementation that can succeed.

Process Primitive	Smaller Team	Larger Team
Life cycle phases	Weak boundaries between phases	Well defined phase synchronize progress among concurrent activities
Artifacts	Focus on technical artifacts, few discrete baselines, very few mgmt. artifacts required.	Change management of technical artifacts which May result in numerous baselines, management artifacts important
Workflow effort allocations	More need for generation, people who perform roles in multiple workflows	Higher percentage of specialist. more people and team focus on specific workflow.
Checkpoints	Many informal events for maintaining technical consistency, no schedule disruption.	A few formal events synchronize among teams which can take days.
Management discipline	Informal planning disruption managed by individual	Formal organisations, formal planning, project control,
Automation discipline	More ad hoc environments managed by individual.	Infrastructure to ensure consistent, up to date environment available access all teams

Table 3.6 Process discriminators that results from differences in project size

Process Primitive	Few stake holders	Multiple stakeholders, Adversarial Relationships
Life cycle phases	Weak boundaries between phases	Well defined phase synchronize progress among concurrent activities
Artifacts	Fewer and less detailed management artifacts required.	Management artifacts paramount, especially the business case.
Workflow effort allocations	Lesser overhead assessment	High assessment overhead to ensure stakeholder concurrence
Checkpoints	Many informal events	3 to 4 formal events.
Management discipline	Informal planning, project control and organisation.	formal planning, project control,
Automation discipline	Insignificant.	Online stake holder environment necessary.

Table 3.7 Process discriminators that results from differences in stakeholder cohesion

Process Primitive	Flexible process	Inflexible process
Life cycle phases	Tolerant to cavalier phase commitments	More credible basis required for inception phase commitments
Artifacts	Changeable business case vision	Carefully controlled changes to business case vision
Workflow effort allocations	Insignificant	Increased levels of management and assessment workflows
Checkpoints	Many informal events for maintaining technical consistency	3 to 4 formal events.
Management discipline	Insignificant	More fidelity required for planning and project cp
Automation discipline	Insignificant.	Insignificant

Table 3.8 Process discriminators that results from differences Process flexibility

Process Primitive	Mature level 3 or 4 organization	Level 1 Organization
Life cycle phases	Well established criteria for phase transitions	Insignificant
Artifacts	Well established format, content and production methods	Free form
Workflow effort allocations	Insignificant	No basis
Checkpoints	Well defined	Insignificant.
Management discipline	Predictable planning, objective status planning	Informal planning and project control.
Automation discipline	Requires high level s of automation for round trip engineering, change management	Little automation

Table 3.9 Process discriminators that results from differences in project maturity

Process Primitive	Complete architecture feasibility demonstration	No architecture feasibility demonstration
Life cycle phases	More inception and elaboration phase iterations	Fewer early iterations/more constructions iterations
Artifacts	Earlier breadth and depth across technical artifacts	Insignificant
Workflow effort allocations	Higher level of design efforts/lower level of implementation and assessment	Higher levels of implementation and assessment to deal.
Checkpoints	More emphasis on executable demonstrations	More emphasis on briefings, documentation and simulation.
Management discipline	Insignificant	Insignificant
Automation discipline	More environment resources required earlier in the life cycle	Less environment demand early in the life cycle.

Table 3.10 Process discriminators that results from differences in architectural risk

Process Primitive	Experienced Team	Inexperienced Team
Life cycle phases	Shorter engineering stage	Longer engineering stage
Artifacts	Less scrape and rework in requirement and design set	More scrap and rework in requirement and design sets
Workflow effort allocations	Lower level of requirement and design	Higher levels of requirements and design
Checkpoints	Insignificant	Insignificant
Management discipline	Less emphasis on risk management/less frequent status assessments needed	More frequent status assessments required
Automation discipline	Insignificant	Insignificant

Table 3.11 Process discriminators that results from differences in domain experience



Stream