SHRIMATI INDIRA GANDHI COLLEGE

Affiliated to Bharathidasan University| Nationally Accredited at 'A' Grade(3rd Cycle) by NAAC

An ISO 9001:2015 Certified Institution

Thiruchirrappalli

STUDY MATERIAL

SOFTWARE ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE, INFORMATION TECHNOLOGY AND COMPUTER APPLICATIONS

Prepared by,

MS.T.R.B.VIDHYA, M.S.I.T.,M.Phil.,M.C.A.,

ASST. PROF. IN COMPUTER SCIENCE,

SHRIMATI INDIRA GANDHI COLLEGE,

TIRUCHIRAPPALLI - 2

Software Engineering

Unit I

Introduction to Software Engineering: Definitions-Size Factors – Quality and Productivity Factors. Planning a Software Project: Planning the Development Process – Planning an Organizational Structure.

Unit II

Software Cost Estimation: Software cost Factors – Software Cost Estimation Techniques –Staffing-Level Estimation – Estimating Software Estimation Costs.

Unit III

Software Requirements Definition: The Software Requirements specification – Formal Specification Techniques. Software Design: Fundamental Design Concepts – Modules and Modularization Criteria.

Unit IV

Design Notations – Design Techniques. Implementation Issues: Structured Coding Techniques – Coding Style – Standards and Guidelines – Documentation Guidelines.

Unit V

Verification and Validation Techniques: Quality Assurance – Walkthroughs and Inspections

Unit Testing and Debugging – System Testing. Software Maintenance: Enhancing Maintainability during Development – Managerial Aspects of Software Maintenance – Configuration Management.

Textbook:

1. Software Engineering Concepts – Richard Fairley, 1997, Tata Mcgraw Hill.

Reference Books:

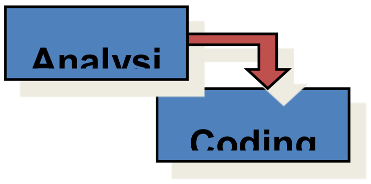
1. Software Engineering for Internet Applications – Eve Anderson, Philip Greenspun, Andrew Grumet, 2006, PHI.
2. Fundamentals of Software Engineering – Rajib Mall, 2nd Edition, PHI
3. Software Engineering – Stephen Schach, 7th edition, TMH.

<u>**UNIT:1**</u>

Software Engineering is the subdiscipline of Computer Science that attempts to apply engineering principles to the creation, operation, modification and maintenance of the software components of various systems. As with much of Computer Science, the subject of Software Engineering is at an very early stage in its development. It is much more of an art than a science, and at present has little in common which classical engineering.

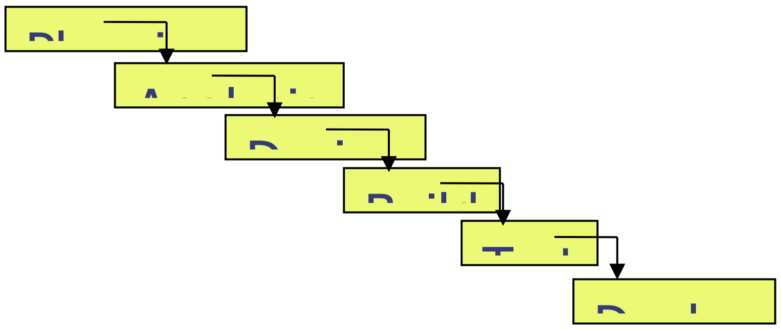
The water fall model

There are two essential steps common to the development of computer programs: <u>analysis and coding.</u>



Both involve creative work that directly contributes to the usefulness of the end product.

In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other 'overhead' steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps."



Waterfall model phases

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
- The drawback of the waterfall model is the difficulty of accommodating change after the process is underway

Waterfall model problems

- Inflexible partitioning of the project into distinct stages
- This makes it difficult to respond to changing customer requirements
- Therefore, this model is only appropriate when the requirements are well-understood

Pros:	Cons:
Easiest to understand	Does not model the real world
Easiest to instrument	Too much documentation
Enforced discipline	
Document and deliverable driven	

Suggested Changes 'Then' and 'Now'

Point 1. "Program design" comes first.

- ➔ Program designer looks at storage, timing, data. Very high level…First glimpse. First concepts…
- During analysis: program designer must then impose storage, timing, and operational constraints to determine consequences.
- Begin design process with program designers, not analysts and programmers
- Design, define, and allocate data processing modes even if wrong. (allocate functions, database design, interfacing, processing modes, i/o processing, operating procedures…. Even if wrong!!)
- ➔ Build an overview document – to gain a basic understanding of system for all stakeholders.

Point 2: Document the Design

- Development efforts required huge amounts of documentation – manuals for everything
  - User manuals; operation manuals, program maintenance manuals, staff user manuals, test manuals…
  - Most of us would like to 'ignore' documentation.
- Each designer MUST communicate with various stakeholders: interface designers, managers, customers, testers, developers, …..

Point 3: Do it twice.

- History argues that the delivered version is really version #2Version 1, major problems and alternatives are addressed – the 'big cookies' such as communications, interfacing, data modeling, platforms, operational constraints, other constraints. Plan to throw first version away sometimes…
- Version 2, is a refinement of version 1 where the major requirements are implemented.
- Version 1 often austere; Version 2 addressed shortcomings!

Point 4: Then: Plan, Control, and Monitor Testing.

- Largest consumer of project resources (manpower, computing time, …) is the test phase.
    - ➔ Phase of greatest risk – in terms of cost and schedule.
    - Occurs last, when alternatives are least available, and expenses are at a maximum.
    - Typically that phase that is shortchanged the most
- **To do:**
    - **1.** Employ a non-vested team of test specialists – not responsible for original design.
    - 2. Employ visual inspections to spot obvious errors (code reviews, other technical reviews and interfaces)
    - 3. Test every logic path
    - 4. Employ final checkout on target computer…..

Point 5 – Old: Involve the Customer

Old advice: involve customer in requirements definition, preliminary software review, preliminary program design (critical design review briefings…)

Now: Involving the customer and all stakeholders is critical to overall project success. Demonstrate increments; solicit feedback; embrace change; cyclic and iterative and evolving software. Address risk early…..

The Conventional Software Management Performance

Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.

a. You can compress software development schedules 25% of nominal, but no more.
b. For every $1 you spend on development, you will spend $2 on maintenance.
c. Software development and maintenance costs are primarily a function of the number of source lines of code.
d. Variations among people account for the biggest differences in software productivity.
e. The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.
f. Only about 15% of software development effort is devoted to programming.
g. Walkthroughs catch 60% of the errors.

h. 80% of the contribution comes from 20% of contributors

The basic parameters of the software cost models

Most software cost models can be abstracted into a function of five basic parameters:

a. Size of the end product which is typically quantified in terms of the number of source instructions or the function points required to develop the required functionality
b. Process used to produce the end product and to avoid non-value adding activities like rework, communication overhead.
c. Personnel particularly their experience with the computer science issues and the applications domain issues of the project.
d. Environment which is made up of the tools and techniques available to support efficient software development and to automate process.
e. Quality of the product, including its performance, reliability, and adaptability.

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

Effort=(Personnel)(Environment)(quality)(size $^{\text{Process}}$)

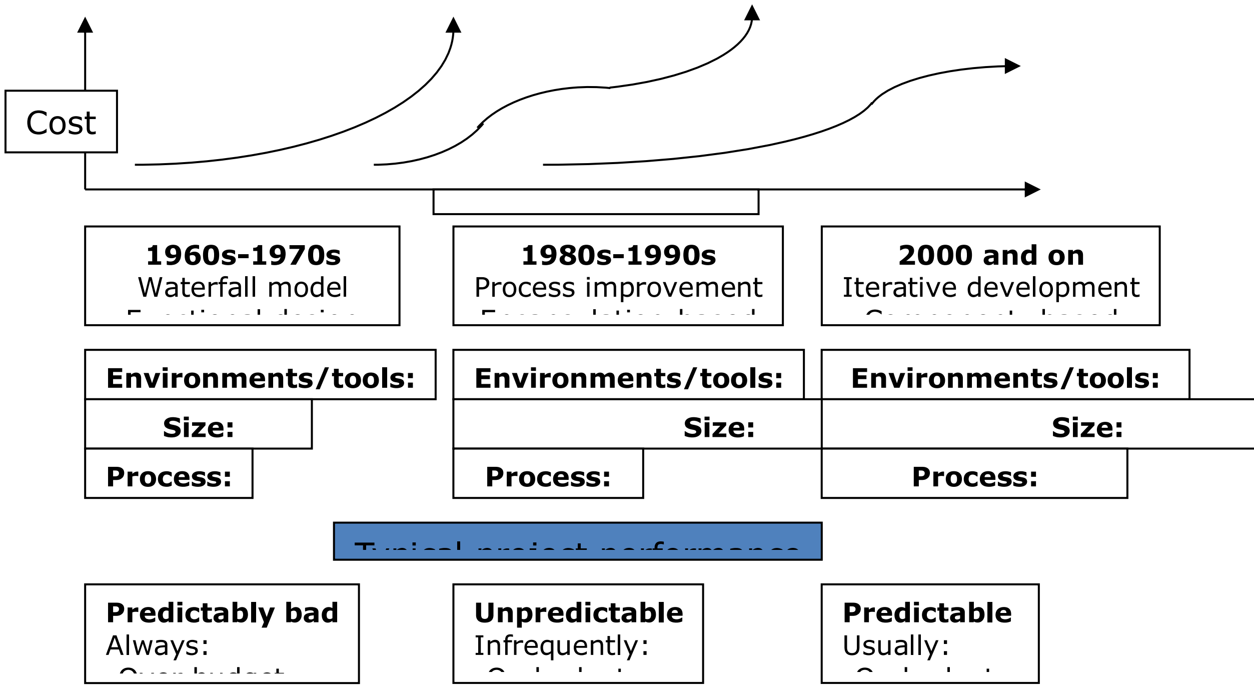
The figure following shows three generations of basic technology advancement in topls, components and process. Thre requited levels of quality and personnel are assumed to be constant. The ordinate of the graph refers to software unit costs like SLOC,Function Point, and component realized by an organization.

The three generations of software development are defined as folows:

Conventional: 1960s and 1970s craftmanship. Organization used custom tools, custom process, and virtually all custom components built in primitive languages.

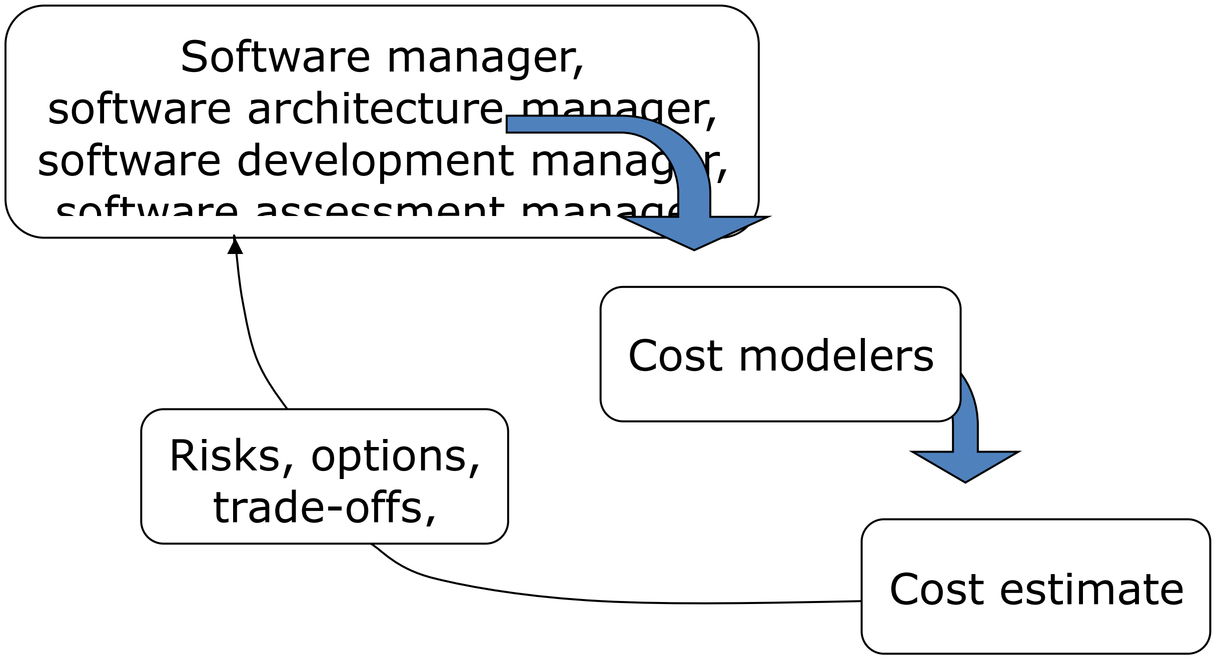
Transition: 1980s and 1990s, software engineering. Organizations used more-repeatable process and off-the-shelf tools and mostly >70% custom components buits in higer level languages. Some of the components <30% were available as commercial products, including the operating system , database management system, networking and graphical user interface.

Modern Pracices: 2000 and late, software production. In this mostly 70% off-the-shelf components perhaps as few as 30% of the components need to be custom built. With advances in software technology and integrated production environments, thse components-based systems can be produced very rapidly.



Three generations of software economics:

The predominant cost estimation process


Software manager,
software architecture manager,
software development manager,
software assessment manager
Cost modelers
Risks, options,
trade-offs,
Cost estimate

- **A good estimate has the following attributes:**

- It is conceived and supported by the project manager, architecture team, development team, and test team accountable for performing the work.
- It is accepted by all stakeholders as ambitious but realizable.
- It is based on a well defined software cost model with a credible basis.
- It is based on a database of relevant project experience that includes similar processes, technologies, environments, quality requirements, and people.
  It is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

To improve the software economics

Five basic parameters of the software cost model:

1. Reducing the size or complexity of what needs to be developed
2. Improving the development process
3. Using more-skilled personnel and better teams
4. Using better environments
5. Trading off or backing off on quality thresholds

1. Reducing Software Product Size

"The most significant way to improve affordability and return on investment is usually to produce a product that achieves the design goals with the minimum amount of human-generated source material."

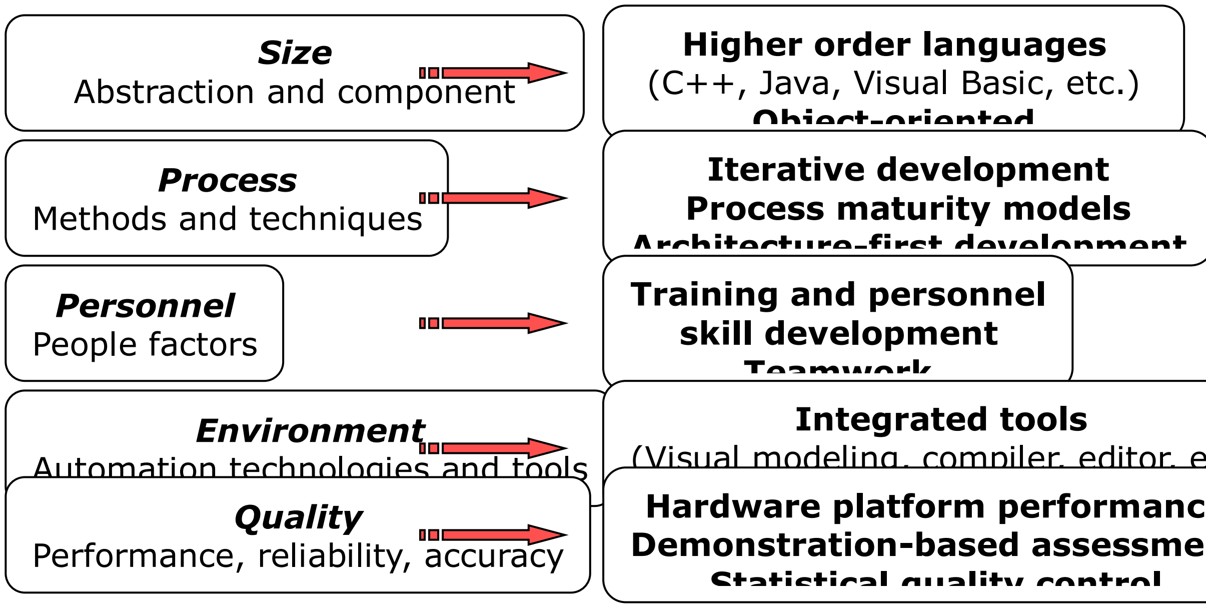
Reuse, object-oriented technology, automatic code production, and higher order programming languages are all focused on achieving a given system with fewer lines of human-specified source directives.

UFP -Universal Function Points

The basic units of the function points are external user inputs, external outputs, internal logic 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



Reducing Software Product Size: Object Oriented

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

Reducing Software Product Size – Reuse

- ☐ 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.
    Reducing Software Product Size – Commercial Components

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

2. Improving Software Processes

Process is an overloaded term. There are three distinct process perspectives.

f. **Meta process:** An Organization's policies, procedures and practices for pursuing a software-intensive line of business

g. **Macro Process:** a project's policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality constraints.

h. **Micro process:** a project team's policies, procedures, and practices for achieving an artifact of the software process.

Three levels of processes and their attributes

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

3. Improving Team Effectiveness

Some rules of team management include the following:

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

In examining how to staff a software project, Boehm offered the following five staffing principles:

- ☐ The principle of top talent: *Use better and fewer people.*
- ☐ The principle of job matching: *Fit the task to the skills an motivation of the people available.*

- ☐ The principle of career progression: *An organization does best in the long run by helping its people to self-actualize.*
- ☐ The principle of team balance: *Select people who will complement and harmonize with one another.*
- ☐ The principle of phase-out: *Keeping a misfit on the team doesn't benefit anyone.*

The following are some attributes of successful software project managers that deserve much more attention (Important Project Manager Skills):

- o 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.
- o Customer-interface skill. Avoiding adversarial relationships among stake-holders is a prerequisite for success.
- o 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.
- o 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.
- o 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

4. Achieving Required Quality

Key practices that improve overall software quality:

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

The Principles of Conventional Software Engineering

1. **Make quality #1**. Quality must be quantified and mechanism 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 an "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model**. Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases**. Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation through-out the life cycle. Why should software engineers use Ada for requirements, design, and code unless Ada were optimal for all these phases?
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 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**. The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success**. Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process will succeed. The wrong people with appropriate tools, languages, and process will probably fail.

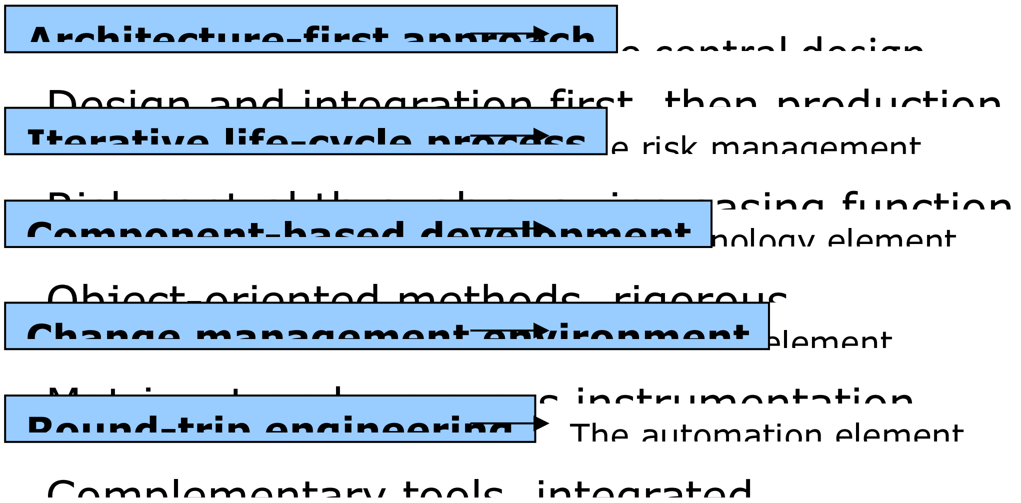
14. **Follow with care**. Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping-all these might increase quality, decrease cost, and increase user satisfaction. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal.
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 design.
16. **Understand the customer's priorities**. It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.
17. **The more they see, the more they need**. The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.
18. **Plan to throw one away** .One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first time.
19. **Design for change**. The architectures, components, and specification techniques you use must accommodate change.
20. **Design without documentation is not design**. I have often heard software engineers say, "I have finished the design. All that is left is the documentation."
21. **Use tools, but be realistic**. Software tools make their users more efficient.
22. **Avoid tricks**. Many programmers love to create programs with tricks- constructs that perform a function correctly, but in an obscure way. Show the world how smart you are by avoiding tricky code.
23. **Encapsulate**. Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.
24. **Use coupling and cohesion**. Coupling and cohesion are the best ways to measure software's inherent maintainability and adaptability.
25. **Use the McCabe complexity measure**. Although there are many metrics available to report the inherent complexity of software, none is as intuitive and easy to use as Tom McCabe's.
26. **Don't test your own software**. Software developers should never be the primary testers of their own software.
27. **Analyze causes for errors**. It is far more cost-effective to reduce the effect of an error by preventing it than it is to find and fix it. One way to do this is to analyze the causes of errors as they are detected.
28. **Realize that software's entropy increases**. Any software system that undergoes continuous change will grow in complexity and become more and more disorganized.

29. **People and time are not interchangeable**. Measuring a project solely by person-months makes little sense.
30. **Expert excellence**. Your employees will do much better if you have high expectations for them.

The Principles of Modern Software Management
Top ten principles:

1. **Base the process on an architecture-first approach:** 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:** Today's sophisticated software systems, it is not possible to define the entire problem, design the entire solution, build the software, the test the end product in sequence. An iterative process that refines the problem understanding, an effective solution, and an effective plan over several iterations encourages a balanced treatment to all objectives.
3. **Component based development:** A component is a cohesive set of preexisting lines of code, either in source or executable format, with a defined interface and behavior.
4. **Change management environment:** The dynamic of iterative development, including concurrent workflows by different teams working on shared artifacts, necessitates objectively controlled baselines.
5. **Round trip engineering:** it is the environment support necessary to automate and synchronize engineering information in different formats. Change freedom a necessity in an iterative process, and establishing an integrated environment is crucial.
6. **Model based notation:** A model based approach supports the evolution of semantically rich graphical and textual design notations
7. **Objective quality control:** It is the best assessment mechanisms are well defined measures derived directly from the evolving engineering
8. **Demonstration based approach:** transitioning the current state of the product artifacts into an executable demonstration of relevant scenarios stimulates earlier convergence on integration
9. **Evolving levels of detail:** the evolution of project increments and generations must be commensurate with the current level of understanding f the requirements and architecture.
10. **Configurable process:** No single process is suitable for all software developments. A pragmatic process framework must be configurable to a broad spectrum of applications.

Waterfall Process	Iterative Process
Requirements first	Architecture first
Custom development	Component based development
Change avoidance	Change management
Ad hoc tools	Round-trip engineering


Architecture-first approach
Design and integration first, then production
Iterative life-cycle process
risk management
Component-based development
Object-oriented methods, rigorous
Change management environment
element
instrumentation
Round-trip engineering
The automation element
Complementary tools, integrated

<u>UNIT-II</u>

1. The Life cycle phases

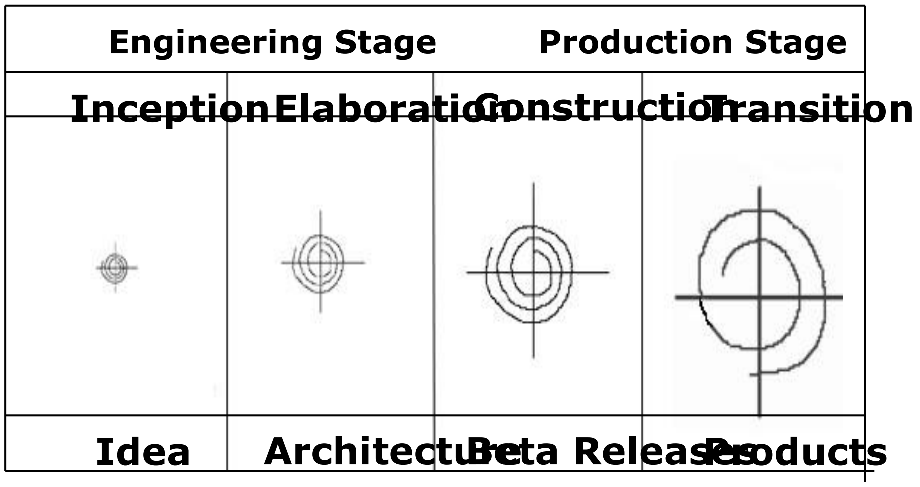
The following are the two stages of the life-cycle:

- *The engineering stage* – driven by smaller teams doing design and synthesis activities
- *The production stage* – driven by larger teams doing construction, test, and deployment activities

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

The engineering stage is decomposed into two distinct phases, inception and elaboration, and the production stage into construction and transition. These four phases of the life-cycle process are loosely mapped to the conceptual framework of the spiral model.

The size of the spiral model corresponds to the inertia of the project with respect to the breadth and depth of the artifacts that have been developed.



In most conventional life cycles, the phases are named after the primary activity within each phase: requirements analysis, design, coding, unit test, integration test, and system test. Conventional software development efforts emphasized a mostly sequential process, in which one activity was required to be complete before the next was begun.

Within an iterative process, each phase includes all the activities, in varying proportions.

Inception Phase:

- ☐ Overriding goal of the inception phase is to achieve concurrence among stakeholders on the life-cycle objectives
- ☐ Essential activities :
    - ➢ *Formulating the scope of the project* (capturing the requirements and operational concept in an information repository)
    - ➢ *Synthesizing the architecture* (design trade-offs, problem space ambiguities, and available solution-space assets are evaluated)
    - ➢ *Planning and preparing a business case* (alternatives for risk management, iteration planes, and cost/schedule/profitability trade-offs are evaluated)

Elaboration Phase:

It is easy to argue that the elaboration phase is the most critical of the four phases. At the end of this phase, the "engineering "is considered complete and the project faces its reckoning. During the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size, risk and novelty of the project.

☐ Essential activities :

- *Elaborating the vision* (establishing a high-fidelity understanding of the critical use cases that drive architectural or planning decisions)
- *Elaborating the process and infrastructure* (establishing the construction process, the tools and process automation support)
- *Elaborating the architecture and selecting components* (lessons learned from these activities may result in redesign of the architecture)

Construction Phase:

☐ During the construction phase :

All remaining components and application features are integrated into the application

All features are thoroughly tested

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

Transition Phase:

- The transition phase is entered when baseline is mature enough to be deployed in the end-user domain. This phase could include beta testing, conversion of operational databases, and training of users and maintainers.
- Essential activities :
    - *Synchronization and integration of concurrent construction into consistent deployment baselines*
    - *Deployment-specific engineering* (commercial packaging and production, field personnel training)
    - *Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set*

Evaluation Criteria:

- Is the user satisfied?

- Are actual resource expenditures versus planned expenditures acceptable?
- Each of the four phases consists of one or more iterations in which some technical capability is produced in demonstrable form and assessed against a set of the criteria.
- The transition from one phase to the nest maps more to a significant business decision than to the completion of specific software activity.

A set represents a complete aspect of the system, an artifact represents cohesive information that typically is developed and reviewed as a single entity.

Life – cycle software artifacts are organized into five distinct sets that are roughly partitioned by the underlying language of the set: management, requirements, design, implementation, and deployment.

Management set:

The Management set captures the artifacts associated with process planning and execution. Management artifacts are evaluated, assessed and measured through a combination of the following:

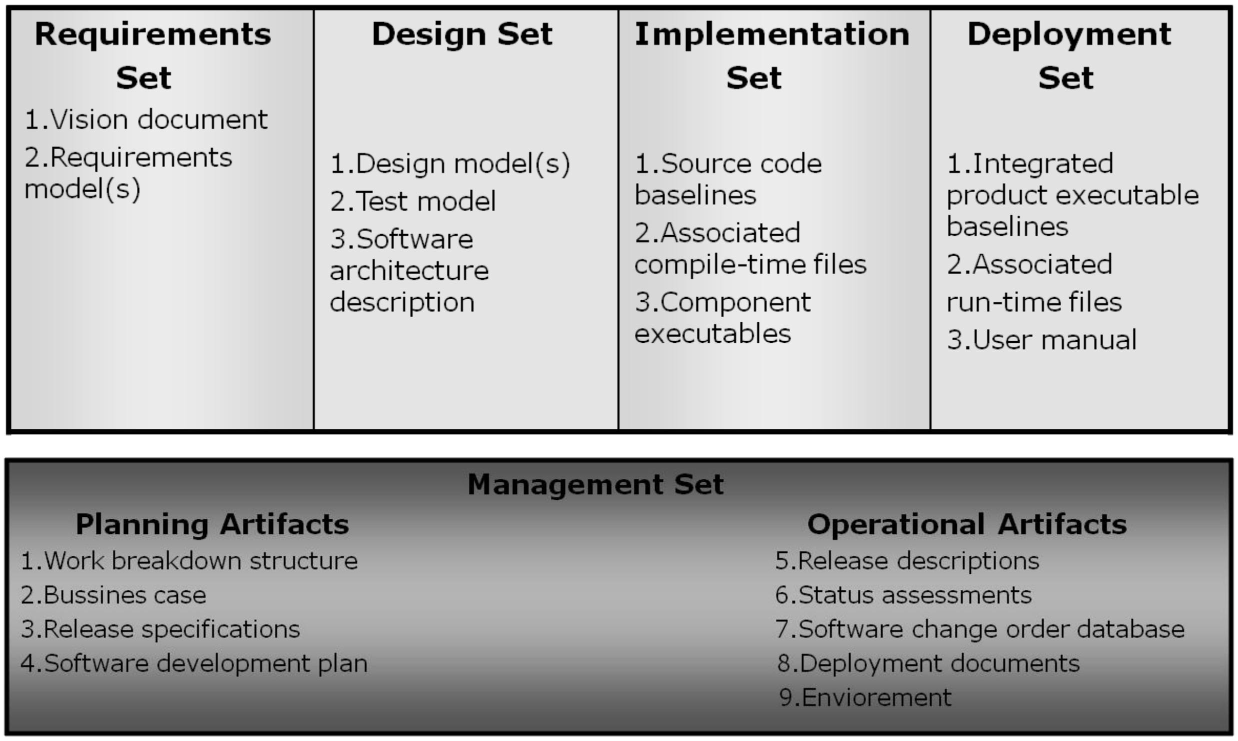
a. Relevant stakeholder reviews

b. Analysis of changes between the current version of the artifact and previous versions

c. Major milestone demonstrations of the balance among all artifacts and in particular, the accuracy of the business case and vision artifacts.

The Engineering Sets:

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



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

Management Set

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

Management artifacts:

The *management* set includes several artifacts

- **Work Breakdown Structure** – vehicle for budgeting and collecting

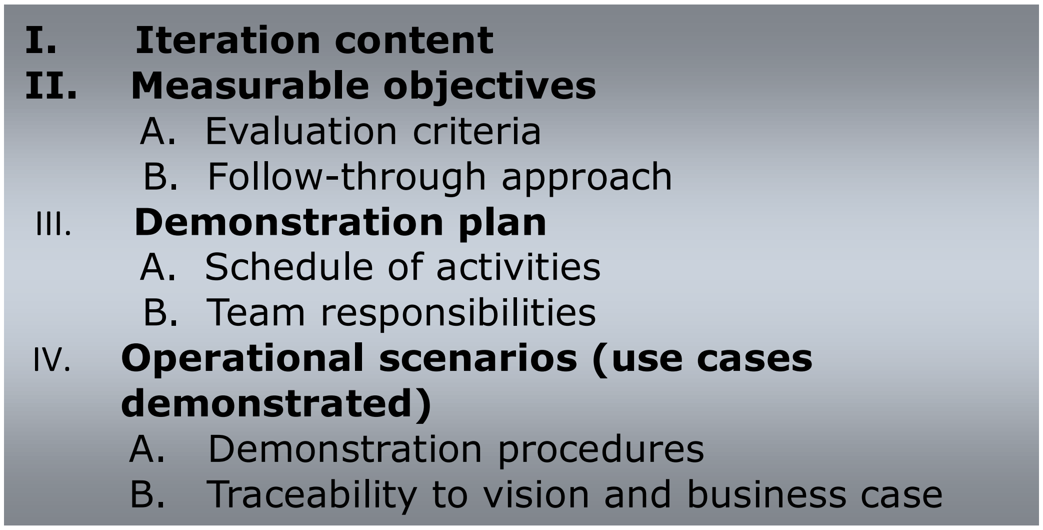
  costs. The software project manager must have insight into project costs

  and how they are expended. If the WBS is structured improperly, it can drive the evolving design in the wrong direction.

- **Business Case** – 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.

  Release Specifications

  <u>Typical release specification outline :</u>



I. Iteration content

II. Measurable objectives

A. Evaluation criteria

B. Follow-through approach

III. Demonstration plan

A. Schedule of activities

B. Team responsibilities

IV. Operational scenarios (use cases demonstrated)

A. Demonstration procedures

B. Traceability to vision and business case

Two important forms of requirements:

- *Vision statement* - which captures the contract between the development group and the buyer.

- *Evaluation criteria* – defined as management-oriented requirements, which may be represented by use cases, use case realizations or structured text representations.

- **Software Development Plan** – the defining document for the project's process. It must comply with the contract, comply with the organization standards, evolve along with the design and requirements.

- **Deployment** – depending on the project, it could include several document subsets for transitioning the product into operational status. It could also include computer system operations manuals, software installation manuals, plans and procedures for cutover etc.

- **Environment** – A robust development environment must support automation of the development process. It should include :

  - requirements management

  - visual modeling

- document automation
- automated regression testing

Engineering Artifacts:

In general review, there are three *engineering* artifacts

- Vision document – supports the contract between the funding authority and

the development organization.

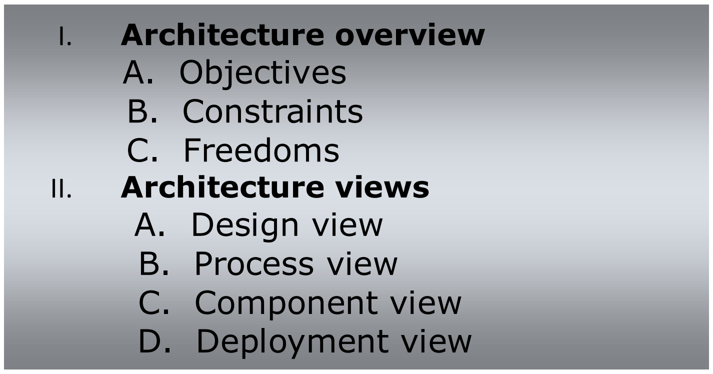
It is written from the user's perspective, focusing on the essential features

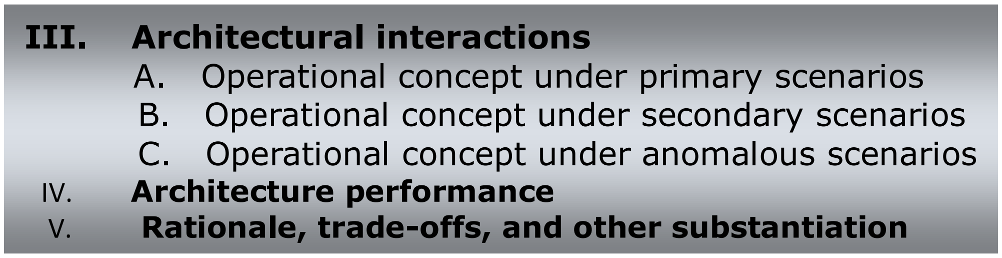
of the system.

It should contain at least two appendixes – the first appendix should describe the operational concept using use cases, the second should describe the change risks inherent in the vision statement.

- **Architecture Description** – it is extracted from the design model and includes views of the design, implementation, and deployment sets sufficient to understand how the operational concept of the requirements set will be achieved.

Typical architecture description outline :


I. **Architecture overview**
   A. Objectives
   B. Constraints
   C. Freedoms
II. **Architecture views**
   A. Design view
   B. Process view
   C. Component view
   D. Deployment view


III. Architectural interactions
   A. Operational concept under primary scenarios
   B. Operational concept under secondary scenarios
   C. Operational concept under anomalous scenarios
IV. **Architecture performance**
V. **Rationale, trade-offs, and other substantiation**

- **Software User Manual** – it should include installation procedures, usage procedures and guidance, operational constraints, and a user interface description.

- It should be written by members of the test team, who are more likely to understand the user's perspective than the development team.
- It also provides a necessary basis for test plans and test cases, and for construction of automated test suites.

Architecture in the management perspective view

From a management perspective, there are three different aspects of an architecture :

- An ***architecture*** **(**the intangible design concept) is the design of software system, as opposed to design of a component.
- An ***architecture baseline*** (the tangible artifacts) is a slice of information across the engineering artifact sets sufficient to satisfy all stakeholders that the vision can be achieved within the parameters of the business case (cost, profit, time, people).
- An ***architecture description*** (a human-readable representation of an architecture) is an organizes subsets of information extracted from the design set model.

The importance of software architecture can be summarized as follows:

- Achieving a stable software architecture represents a significant project milestone at which the critical make/buy decisions should have been resolved.
- Architecture representations provide a basis for balancing the trade-offs between the problem space and the solution space.
- The architecture and process encapsulate many of the important communications among individuals, teams, organizations, and stakeholders.
- Poor architectures and immature processes are often given as reasons for project failures.
- A mature process, an understanding of the primary requirements, and a demonstrable architecture are important prerequisites for predictable planning.
- Architecture development and process definition are the intellectual steps that map the problem to a solution without violating the constraints.

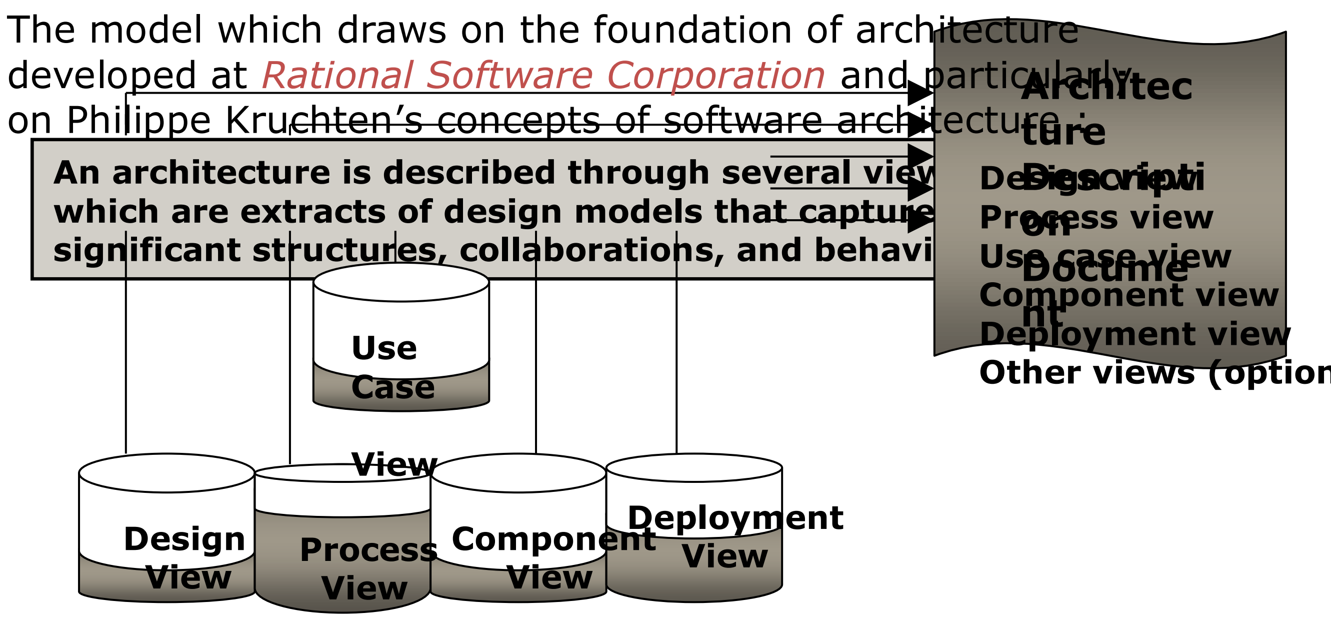
Architecture in the technical perspective view

An architecture framework is defined in terms of views that are abstractions of the UML models in the design set. The design model includes the full breadth and depth of information.

An architecture view is an abstraction of the design model, it contains only the architecturally significant information.

Most real world systems require four views: design, process, component, and deployment. The purposes of these views are as follows:

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


The model which draws on the foundation of architecture developed at Rational Software Corporation and particularly on Philippe Kruchten's concepts of software architecture :
An architecture is described through several views which are extracts of design models that capture significant structures, collaborations, and behavi
Architecture Description Document
Design view
Process view
Use case view
Component view
Deployment view
Other views (option
Use Case View
Design View
Process View
Component View
Deployment View

➢ The ***use case view*** describes how the system's critical use cases are realized by elements of the design model. It is modeled statically using case diagrams, and dynamically using any of the UML behavioral diagrams. The ***design view*** addresses the basic structure and the functionality of the solution.

➢ The ***process view*** addresses the run-time collaboration issues involved in executing the architecture on a distributed deployment model, including the logical software network topology, interprocess communicationand state management.

➢ The ***component view*** describes the architecturally significant elements of the implementation set and addresses the software source code realization of the system from perspective of the project's integrators and developers.

➢ The ***deployment view*** addresses the executable realization of the system, including the allocation of logical processes in the distribution view to physical resources of the deployment network.

Generally an architecture baseline should include the following:

- Requirements: critical use cases, system level quality objectives, and priority relationship among features and qualities
- Design: names, attributes, structures, behaviors, groupings and relationships of significant classes and components
- Implementation: source component inventory and bill of materials (number, name, purpose, cost) of all primitive components
- Deployment: executable components sufficient to demonstrate the critical use cases and the risk associated with achieving the system qualities

Work flow and software process workflows

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

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

Four basic key principles of the modern process frame work:

1. ***Architecture-first approach*****:** implementing and testing the architecture must precede full-scale development and testing and must precede the downstream focus on completeness and quality of the product features.
2. ***Iterative life-cycle process*****:** the activities and artifacts of any given workflow may require more than one pass to achieve adequate results.
3. ***Roundtrip engineering*****:** Raising the environment activities to a first-class workflow is critical; the environment is the tangible embodiment of the project's process and notations for producing the artifacts.
4. ***Demonstration-based approach*****:** Implementation and assessment activities are initiated nearly in the life-cycle, reflecting the emphasis on constructing executable subsets of the involving architecture.

The iteration workflows of the software process

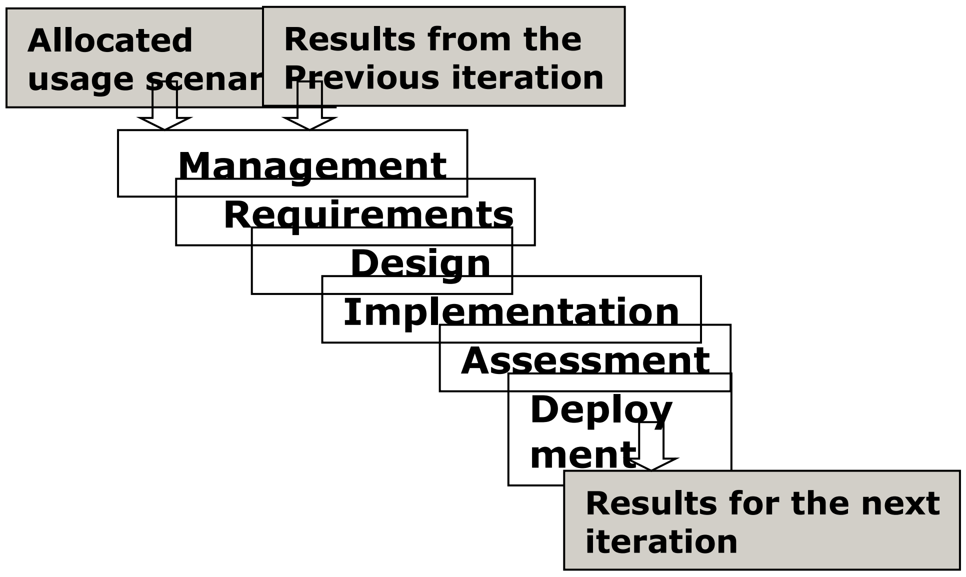
Iteration consists of sequential set of activities in various proportions, depending on where the iteration is located in the development cycle. Each iteration is defined in terms of a se t of allocated usage scenarios. The components needed to implement all selected scenarios are developed and integrated with the results of previous iterations. An individual iteration's workflow illustrated in the following sequence:

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

- Environment: evolving the software change order database to reflect all new baselines and changes to existing baselines for all product, test and environment components

- Requirements: analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to the demonstrated at the end of the iteration and their evaluation criteria.

- Design: Evolving the baseline architecture ad the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate against the evolution criteria allocated to this iteration.

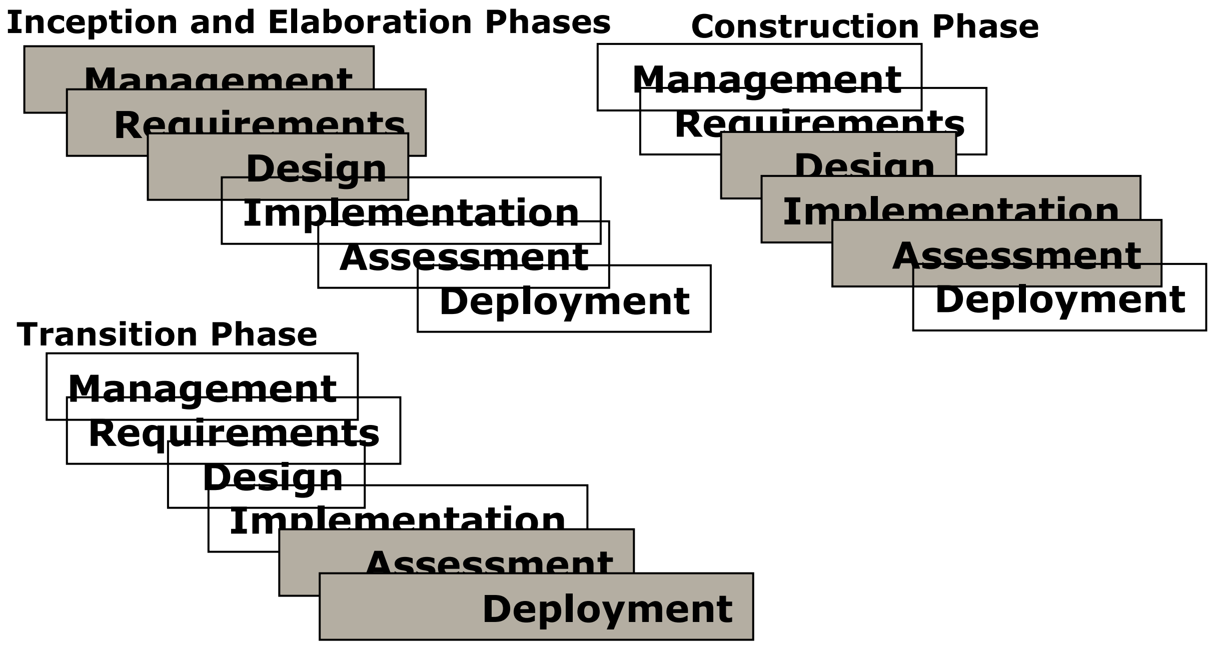


- Implementation: developing any new components, and enhancing or modifying any existing components, to demonstrate the evolution criteria allocated to this iteration
- Assessment: evaluating the results of the iteration, including compliance with the allocated evaluation criteria and the quality of the current baselines; indentifying

any rework required and determining whether it should be performed before deployment of this release or allocated to the next release.

- Deployment: transitioning the released either to an external organization or to internal closure by conducting a post mortem so that lessons learned can be captured and reflected in the next iteration.

The following is an example of a simple development life cycle, illustrates the difference between iterations and increments. This example also illustrates a typical build sequence from the perspective of an abstract layered architecture.



Iteration emphasis across the life cycle

It is important to have visible milestones in the life cycle , where various stakeholders meet to discuss progress and planes. The purpose of this events is to:

- Synchronize stakeholder expectations and achieve concurrence on the requirements, the design, and the plan.
- Synchronize related artifacts into a consistent and balanced state.
- Synchronize related artifacts into a consistent and balanced state Identify the important risks, issues, and out-of-tolerance conditions.
- Perform a global assessment for the whole life-cycle.

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

1. **Major milestones** –provide visibility to system wide issues, synchronize the management and engineering perspectives and verify that the aims of the phase have been achieved.
2. **Minor milestones** – iteration-focused events, conducted to review the content of iteration in detail and to authorize continued work.
3. **Status assessments** – periodic events provide management with frequent and regular insight into the progress being made.

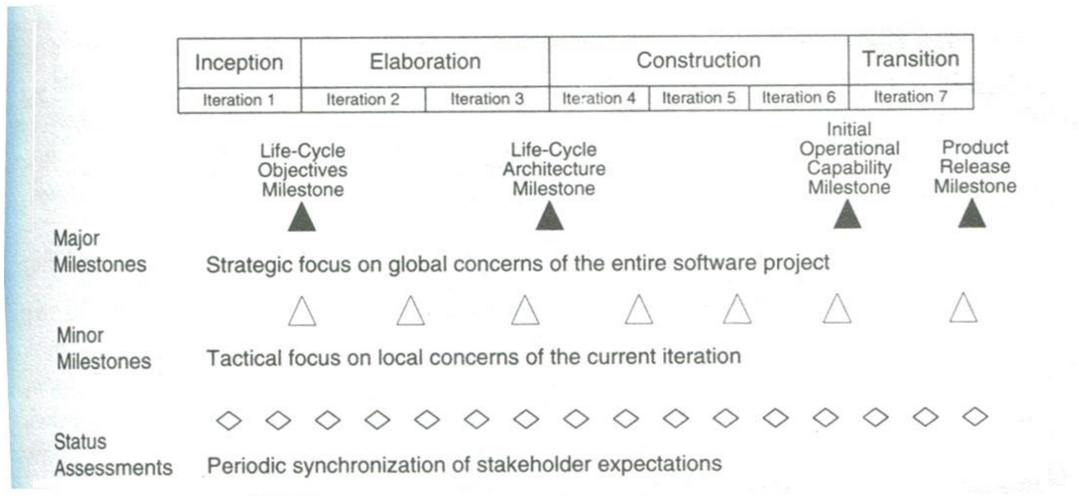

Inception
Elaboration
Construction
Transition
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Life-Cycle Objectives Milestone
Life-Cycle Architecture Milestone
Initial Operational Capability Milestone
Product Release Milestone
Major Milestones
Strategic focus on global concerns of the entire software project
Minor Milestones
Tactical focus on local concerns of the current iteration
Status Assessments
Periodic synchronization of stakeholder expectations

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

<u>MAJOR MILESTONES:</u>

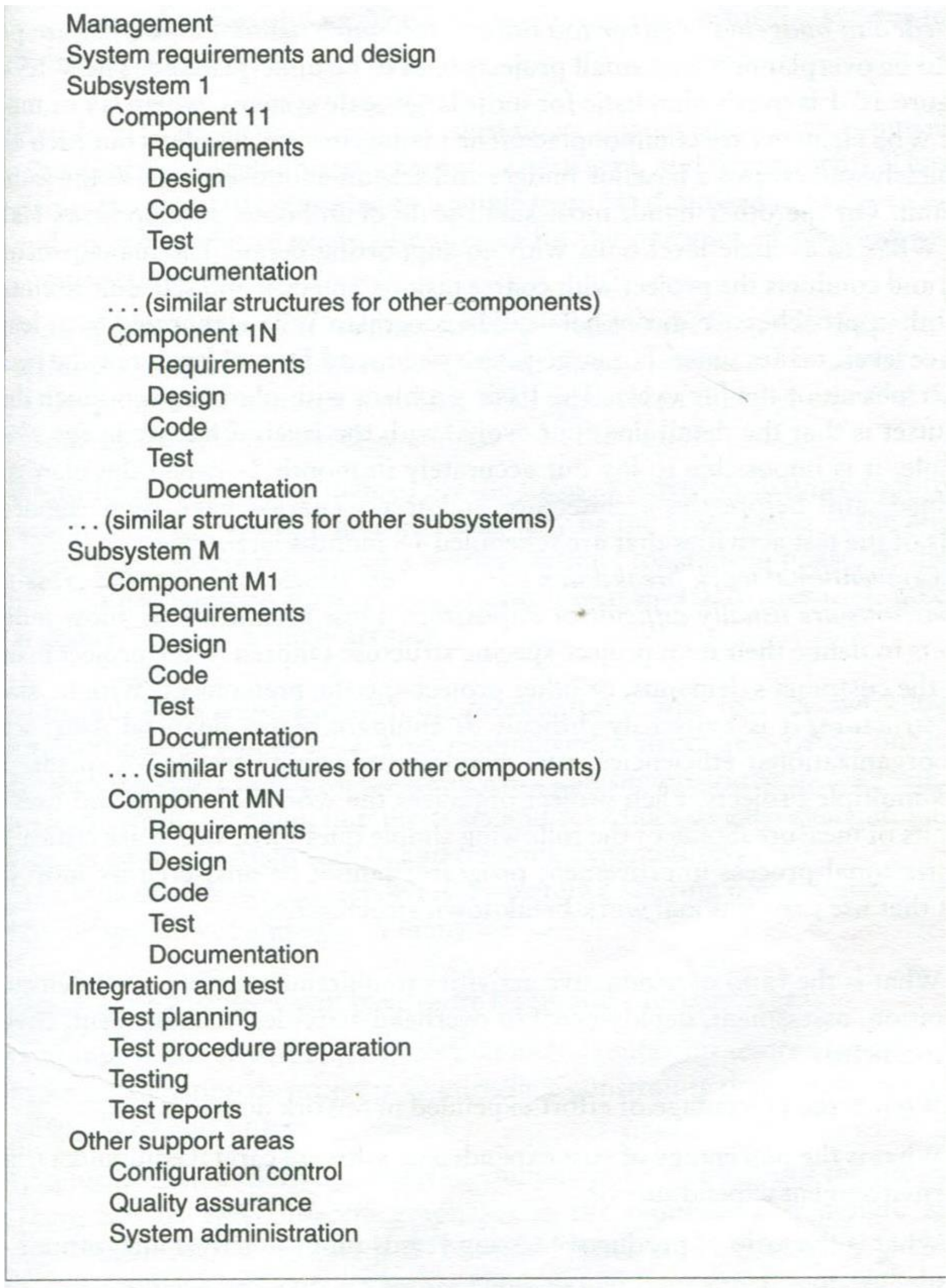
The four major milestones occur at the transition points between life-cycle phases. They can be used in many different process models, including the conventional waterfall model. In an iterative model, the major milestones are used to achieve concurrence among all stakeholders on the current state of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

- Customers: schedule and budget estimates, feasibility , risk assessment, requirements understanding, progress, product line compatibility
- Users: consistency with requirements and usage scenarios, potential for accommodating growth, quality attributes.
- Architectures 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 usuage scenario descriptions, frameworks for component selection of development, resolution of development risk, sufficiency of the development environment
- Maintainers: sufficiency of product and documentation artifacts, understandability, interoperability with existing systems, sufficiency of maintenance environment.

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

The milestones may be conducted as one continuous meeting of all concerned parties or incrementally through mostly on-line review of the various artifacts. There are considerable differences in the levels of ceremony for these events depending on several factors.

The essence of each major milestone is to ensure that the requirements understanding, the life-cycle plans, and the product's form, function, and quality are evolving in balanced levels of detail and to ensure consistency among the various artifacts. The following table summarizes the balance of information across the major milestones.



- Management
- System requirements and design
- Subsystem 1
  - Component 11
    - Requirements
    - Design
    - Code
    - Test
    - Documentation
  - . . . (similar structures for other components)
  - Component 1N
    - Requirements
    - Design
    - Code
    - Test
    - Documentation
- . . . (similar structures for other subsystems)
- Subsystem M
  - Component M1
    - Requirements
    - Design
    - Code
    - Test
    - Documentation
  - . . . (similar structures for other components)
  - Component MN
    - Requirements
    - Design
    - Code
    - Test
    - Documentation
- Integration and test
  - Test planning
  - Test procedure preparation
  - Testing
  - Test reports
- Other support areas
  - Configuration control
  - Quality assurance
  - System administration

FIGURE 10-1. *Conventional work breakdown structure, following the product hierarchy*

MINOR MILESTONES:

All iterations are not created equal. An iteration can take on very different forms and priorities, depending on where the project is in the life cycle. Early iterations focus on analysis and design with substantial elements of discovery, experimentation, and risk assessment. Later iterations focus much more 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 the evolution 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 of which the iteration achieved its objectives and satisfied its evaluation criteria, to review iteration achieved its objectives and satisfied its evaluation criteria, to review iteration results, to review qualification test results, to determine the amount of rework to be done, and to review the impact of the iteration results on the plan for subsequent iterations.

PERIODIC STATUS ASSESSMENTS:

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

Status assessments provide the following:

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

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

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

  The default content of periodic status assessments should include the topics indentified in the following table.

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

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

FIGURE 10-2. *Default work breakdown structure*

UNIT – III & UNIT- IV

- ❑ A WBS is simply a hierarchy of elements that decomposes the project plan into the discrete work tasks. A WBS provides the following information structure:
  - ➢ A delineation of all significant work
  - ➢ A clear task decomposition for assignment of responsibilities
  - ➢ A framework for scheduling, budgeting, and expenditure tracking.
- ❑ 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.

I. Conventional WBS Issues:

Conventional WBS frequently suffer from three fundamental flaws:

1. Conventional **WBS** are prematurely structured around the product design:

   The figure following shows the typical conventional WBS that has been structured primarily around subsystems of its product architecture, the further decomposed into the components of each subsystem.

   Once this structure is ingrained in the WBS and then allocated to responsible managers with budgets, schedules and expected deliverables, a concrete planning foundation has been set that is difficult and expensive to change.

   (SCAN FIG1)

2. Conventional **WBS** are prematurely decomposed, planned, and budgeted in wither too much or too little detail:

   Large software projects tend to be over planned and small projects tend to be under planned. The WBS shown in the above figure is overly simplistic for most large scale systems, where size or more levels of WBS elements are commonplace.

3. Conventional **WBS** are project-specific, and cross-project comparisons are usually difficult or impossible:

   Most organizations allow individual projects to define their own project-specific structure tailored to the project manager’s style, the customer’s demands, or other project-specific preferences.

It is extremely difficult to compare plans, financial data, schedule data, organizational efficiencies, cost trends, productivity tends, or quality tends across multiple projects.

Some of the following simple questions, which are critical to any organizational process improvement program, cannot be answered by most project teams that use conventional WBS.

- What is the ratio of productive activities to overhead activities?
- What is the percentage of effort expanded in rework activities?
- What is the percentage of cost expended in software capital equipment
- What is the ration of productive testing versus integration?
- What is the cost of release?

II. Evolutionary Work Breakdown Structures:

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

- First level WBS elements are the workflows(Management, environment, requirement, design, implementation, assessment, and deployment)
- Second level elements are defined for each phase of the life cycle(inceptions, elaboration, construction and transition)
- Third level elements are defined for the focus of activities that produce the artifacts of each phase.

A default WBS consistent with the process framework (phases, workflows, and artifacts) is shown in the following figure

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

- Scale: Larger projects will have more levels and substructures.
- Organizational structure: projects that include subcontractors or span multiple organizational entities may introduce constraints that necessitate different WBS allocations.
- Degree of custom development: depending on the character of the project there can be very different emphases in the requirements, design, and

implementation workflows. A business process re-engineering project based primarily on existing components would have much more depth in the requirements elements and a fairly shallow design and implementation element.

- Business context: contractual projects require much more elaborate management and assessment elements. Projects developing commercial products for delivery to a board customer base may require much more elaborate substructures for the deployment element.

- Precedent experience: very few projects start with a clean state. Most of them are developed as new generations of a legacy system or in the context of existing organizational standards. It is important to accommodate these constraints to ensure that new projects exploit the existing experience base and benchmarks of project performance.

Planning guidelines

Software projects span a board range of application domains. It is valuable but risky to make specific planning recommendations independent of project context.

Project independent planning advice is also risky. There is the risk that the guidelines may be adopted blindly without being adapted to specific project circumstances. There is also the risk of misinterpretation.

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

- The first guideline prescribes a default allocation of costs among the first-level WBS elements.
- The second guideline prescribes the allocation of effort and schedule across the life cycle phases.

Given an initial estimate of total project cost and these two tables, developing a staffing profile, and allocation of staff resources to reams, a top-level project schedule, and an initial WBS with task budgets and schedules is relatively straightforward.

FIRST-LEVEL WBS ELEMENT	DEFAULT BUDGET
Management	**10%**
Environment	**10%**

Requirements	**10%**
Design	**15%**
Implementation	**25%**
Assessment	**25%**
Deployment	**5%**
Total	**100%**

The first guideline prescribes a default allocation of costs among the first-level WBS elements

The above table provides default allocations for budgeted costs of each first-level WBS element. While these values are certain to vary across projects, this allocation provides a good benchmark for assessing the plan by understanding the rationale for deviations from these guidelines. An important point here is that this is cost allocation, not effort allocation. To avoid misinterpretation two explanations are necessary

1. The cost of different labor categories is inherent in these numbers
2. The cost of hardware and software assets that support the process automation and development teams is also included in the environment element.

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

The second guideline prescribes the allocation of effort and schedule across the life-cycle phases

The above table provides guidelines for allocating effort and schedule across the life cycle phases. Although these values can also vary widely depending on the specific constraints of an application, they provide an average expectation across a spectrum of application domains.

The cost and schedule estimating process

Project plans need to be derived from two perspectives.

The first is a forward-looking top-down approach. It starts with as understanding of the general requirements and constraints, derives a macro –level budgets and intermediate milestones .

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.

The second perspective is a backward-looking, bottom-up approach. We start with the end in mind, analyze the micro-level budgets and schedules, the sum all these elements into the higher level budgets and intermediate milestones.

Backward-looking:

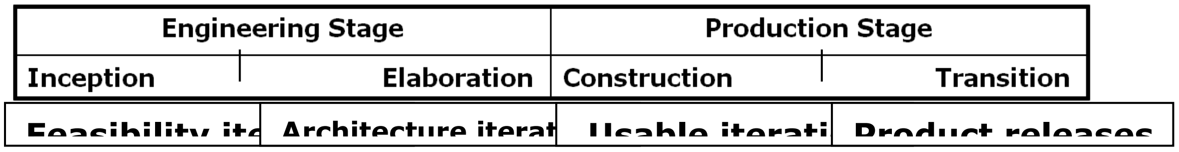
1. The lowest level WBS elements are elaborated into detailed tasks, for which budgets and schedules are estimated by the responsible WBS element manager.

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.

These two planning approaches should be used together, in balance, throughout the life cycle of the project. During the engineering stage, the top-down perspective will dominate. During the production stage, these should be enough precedent experience and planning fidelity that the bottom up planning perspective will dominate.

By then, the top-down approach should be well tuned to the project specific parameters, so is should be used more as a global assessment technique. The following figure shows this life cycle planning balance.



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

The types of project organizations have

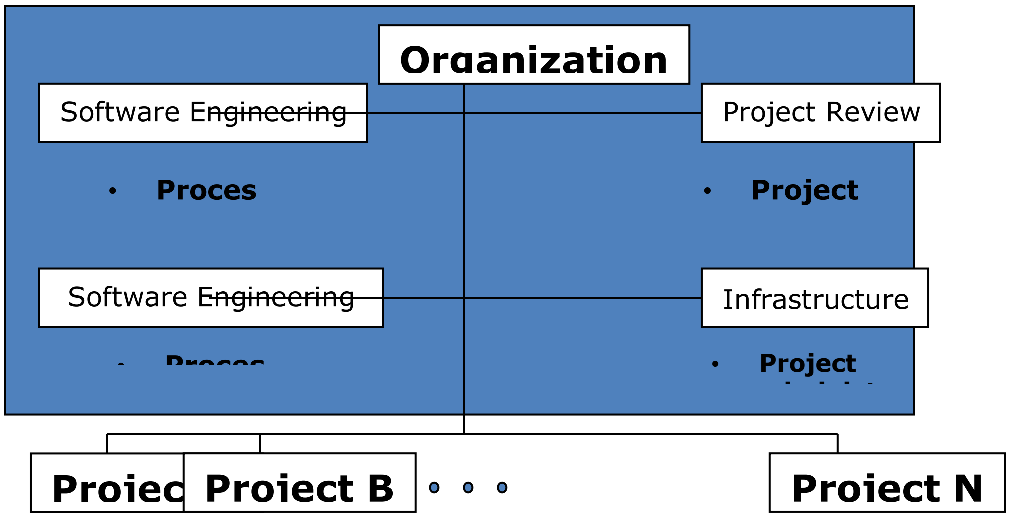
Line of Business Organizations:

Maps roles and responsibilities to a default line-of-business organization. This structure can be tailored to specific circumstances.

The main features of the default organization are as follows:

- Responsibility for process definition and maintenance is specific toa cohensive line of business where process commonality makes sense. For example the process of developing avionics software is different from the process used to develop office applications.

- Responsibility for process automation, is an organizational role and its equal in importance to the process definition role.
- Organizational role may be fulfilled by a single individual or several different teams, depending on the scale of the organization.



Software Engineering Process Authority:

The Software Engineering Process Authority (SEPA) facilitates the exchange of information and process guidance both to and from project practitioners. The SEBA must help initiate and periodically assess project process. The SEBA is necessary role in any organization. The SEBA could be a single individual, the general manager, or even a team of representatives. The SEBA must truly be an authority, competent and powerful.

Project Review Authority:

The project review authority (PRA) is the single individual responsible for ensuring that a software project complies with all organizational and business unit software policies, practices, and standards. The PRA reviews both the project's conformance to contractual obligations and the project's organizational policy obligations. The customer monitors contract requirements, contract milestones, contract deliverable, monthly management reviews, progress, quality, cost, schedule and risk. The PRA reviews customer commitments as well as adherence to organizational policies, organizational deliverables, and financial performance and other risks and accomplishments.

Software Engineering Environment Authority:

The Software Engineering Environment Authority (SEEA) is responsible for automating the organizations process, maintaining the organizations standard environment, training project to use environment, and maintain organization-wide reusable assets. The SEEA rule is necessary to achieve significant return on investment for a common process.

Infrastructure:

An organization infrastructure provides human resource support, project-independent research and development, and other capital software engineering assets. The typical components of the organizational infrastructure are as follows

- **Project Administration:**

  Time accounting system; contracts, pricing, terms and condition; corporate information system integration.

- **Engineering Skill Centers:**

  Custom tools repository and maintenance, bid and proposal support, ind3ependent research and development.

- **Professional Development:**

  Internal training boot camp, personnel recruiting, personnel skills database maintenance, literature and assets library, technical publications.

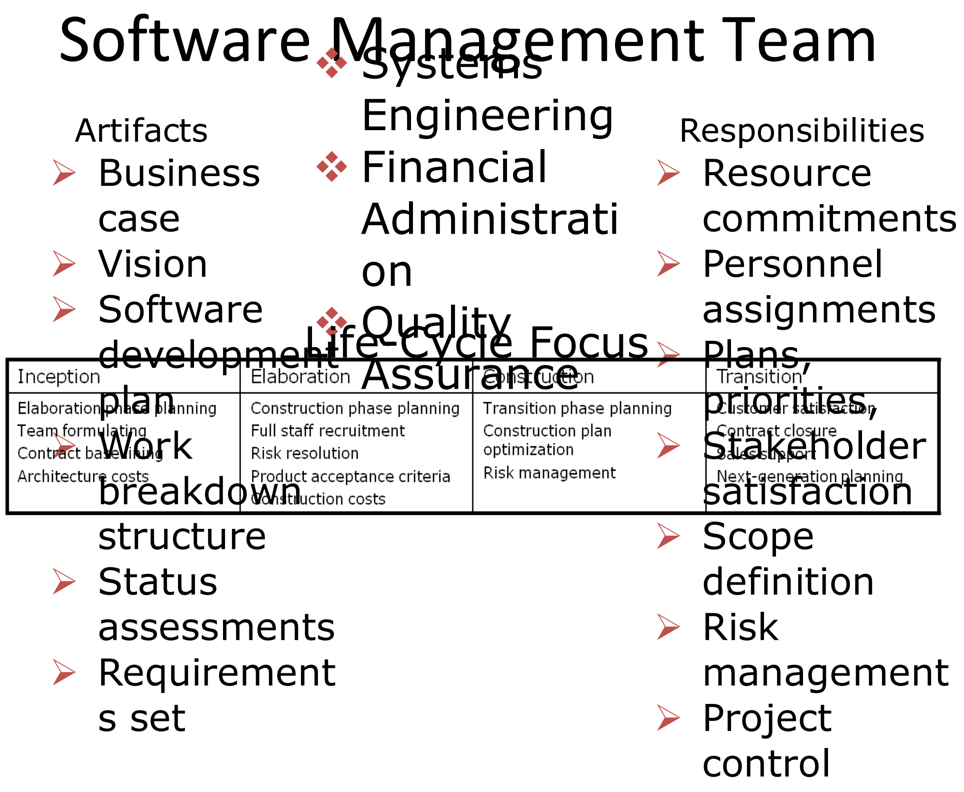
Project Organizations and how it handle their teams?

A default project organization and maps project-level roles and responsibilities. The structure can be tailored to the size and circumstances of the specific project organization. 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 function.
- The development team owns the component construction and maintenance activities.
- Quality is everyone job, integrated into all activities and check points. Each team take responsibility for a different quality perspective.

Software Management Team:

Most project are over constrained. Schedules, cost, functionality and quality expectations are highly inter related and require continuous negotiation among multiple stake holders who have different goals. The software management team carries the burden of delivering with condition to all stake holders. The software management team takes ownership of all aspects of quality.



Software Management Team

Artifacts
- Business case
- Vision
- Software development plan
- Work breakdown structure
- Status assessments
- Requirements set

- Systems Engineering
- Financial Administration
- Quality Assurance

Responsibilities
- Resource commitments
- Personnel assignments
- Plans, priorities,
- Stakeholder satisfaction
- Scope definition
- Risk management
- Project control

Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Elaboration phase planning Team formulating Contract baselining Architecture costs	Construction phase planning Full staff recruitment Risk resolution Product acceptance criteria Construction costs	Transition phase planning Construction plan optimization Risk management	Customer satisfaction Contract closure Sales support Next-generation planning

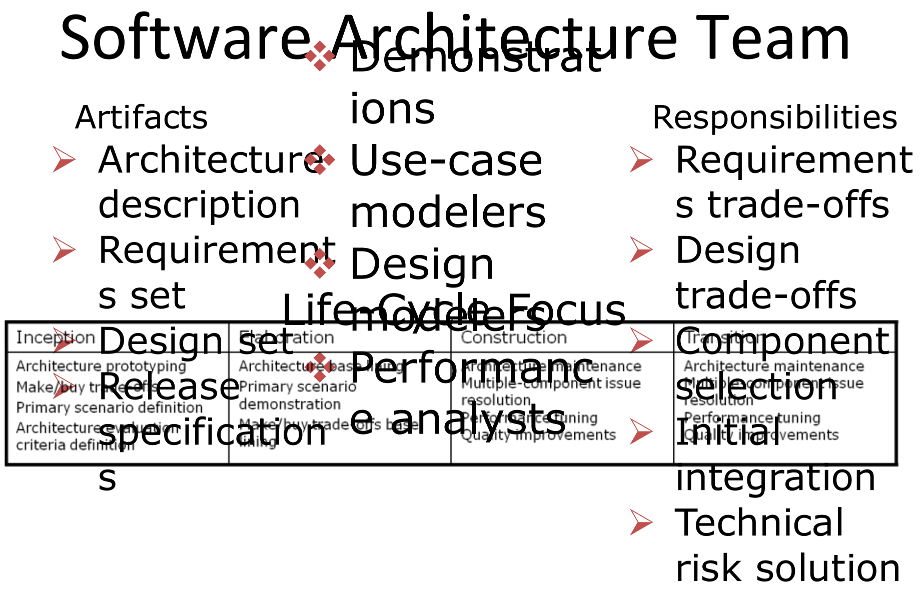
Software Architecture Team:

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

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

- Domain experience to produce an acceptable design view (architecturally significant element s of the design model) and use case view (architecturally significant element s of the use case model).

- ➢ Software technology experience to produce an acceptable process view(concurrency and control thread relationships among the design, component, and deployment models), component view (structure of the implementation set), and deployment view(structure of the deployment set).



Software Architecture Team

Artifacts
- ➢ Architecture description
- ➢ Requirements set
- ➢ Design set
- ➢ Release specifications

- ❖ Demonstrations
- ❖ Use-case modelers
- ❖ Design modelers
- ❖ Performance analysts

Responsibilities
- ➢ Requirements trade-offs
- ➢ Design trade-offs
- ➢ Component selection
- ➢ Initial integration
- ➢ Technical risk solution

Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Architecture prototyping Make/buy trade-offs Primary scenario definition Architecture evaluation criteria definition	Architecture base-lining Primary scenario demonstration Make/buy trade-offs base-lining	Architecture maintenance Multiple-component issue resolution Performance tuning Quality improvements	Architecture maintenance Multiple-component issue resolution Performance tuning Quality improvements

Software development team:

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

- ➢ Commercial component:

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

Software Development Team

❖ Component teams

Artifacts

- Design set
- Implementation set
- Deployment set

Responsibilities

- Component design
- Component implementation
- Component stand-alone test
- Component maintenance
- Component documentation

Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Prototyping support Make/buy trade-offs	Critical component design Critical component implementation and test Critical component base line	Component design Component implementation Component stand-alone test Component maintenance	Component maintenance Component documentation

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

Software Assessment Team:

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

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

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

Software Assessment Team

Artifacts

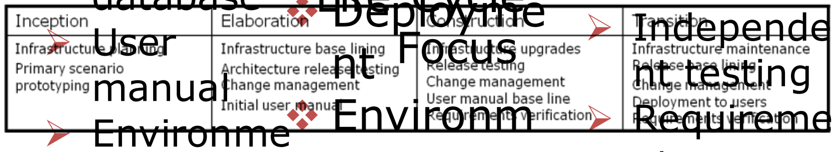
- Deployment set
- SCO database
- User manual
- Environment
- Release specifications
- Release descriptions
- Deployment documents

- Release testing
- Change management
- Deployment
- Environment support

Responsibilities

- Project infrastructure
- Independent testing
- Requirements verification
- Metrics analysis
- Configuration control
- Change management
- User deployment

Life-Cycle Focus



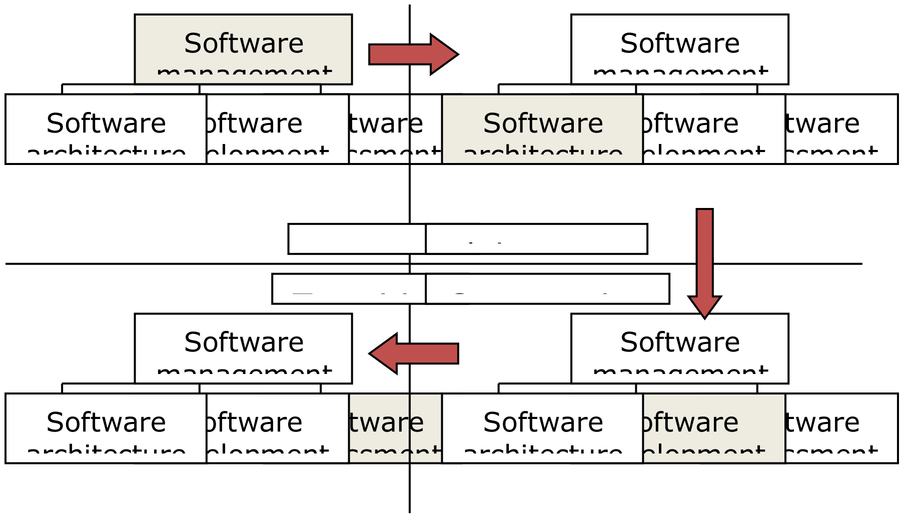
Inception	Elaboration	Construction	Transition
Infrastructure planning Primary scenario prototyping	Infrastructure base lining Architecture release testing Change management Initial user manual	Infrastructure upgrades Release testing Change management User manual base line Requirements verification	Infrastructure maintenance Release base lining Change management Deployment to users Requirements verification

The evaluation of organizations

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

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

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



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

The automation tools available for building blocks in software process

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, as illustrated in the following figure. Each of the process workflows has a distinct for automation support.

Management: there are many opportunities for automating the project planning and control activities of the management work flows. Software cost estimating tools and WBS tools are usual for generating the planning artifacts.

Environment: configuration management and version control are essential in a modern interactive development process

Requirements: conventional approaches decomposed system requirements into subsystems requirements, subsystem requirements into component requirement and component requirements into unit requirements. In a modern project the system requirements are captured in the vision statement. Lower levels of requirements are driven the process organized by iteration rather than by lower level component.

Design: the primary support required for the design work flow is visual modeling, which is used for capturing design models, presenting them in human readable format, and translating them into source code. An architecture first and demonstration based process is enabled by existing architecture components and middle ware.

Implementation: the implementation work flow relies primarily on a programming environment but must also include substantial integration with the change management tools, visual modeling tools, and test automation tools to support productive iteration.

Assessment and deployment: the assessment workflows require all the tools just discussed as well as additional capabilities to support test automation and test management. Defect tracking is another important that supports assessment.

The metrics for managing a modern process

Many different metrics may be of value in managing a modern process. There are seven core metrics that should be used on all software projects. Three are management indicators and four are quality indicators.

Management indicators:

1. Work and progress
2. Budgeted cost and expenditure
3. Staffing team dynamics

Quality indicators:

1. Change traffic and stability
2. Breakage and modularity
3. Rework and adaptability
4. Mean time between failure and maturity

METRIC	PURPOSE	PERSPECTIVES
Work and progress	**Iteration planning, plan vs. actualS, management indicator**	**SLOC, function points, object points, scenarios, test cases, SCOs**
Budget cost and expenditures	**Financial insight, plan vs. actualS, management indicator**	**Cost per month, full-time staff per month, percentage of budget expended**

Staffing and team dynamics	**Resource plan vs. actuals, hiring rate, attrition rate**	**People per month added, people per month leaving**
Change traffic and stability	**Iteration planning, management indicator of schedule convergence**	**Software changes**
Breakage and modularity	**Convergence, software scrap, quality indicator**	**Reworked SLOC per change, by type, by release/component/subsystem**
Rework and adoptability	**Convergence, software rework, quality indicator**	**Average hours per change, by type, by release/component/subsystem**

small scales projects Vs large scale projects

The lists elaborate some of the key differences in discriminators of success. None of these process components is unimportant although some of them are more important than others. ‘

- Design is key in both domains. Good design of a commercial product is a key differentiator in the market place and is the foundation for efficient new product releases.
- Management is paramount in large projects where the consequences of planning errors , resource allocation errors, inconsistent stake holder expectation, and other out of banlaned factors we can have catastrophic consequences for the overall team dynamics
- Deployment plays a far greater role for a small commercial product because there is a broad user base of diverse individuals and environments

Rank	**Large Complex Project**	**Small Commercial Project**
1	Management	Design
2	Design	Implementation
3	Requirements	Deployment
4	Assessment	Requirements
5	Environment	Assessments

6	Implementation	Management
7	Deployment	Environment

UNIT -V

Risk:

"An uncertain event or condition that, if it occurs, has a positive or negative effect on a project's objectives. "

"The chance of exposure to the adverse consequences of future event. "

People may different terms but a key elements of a risk follow

- It relates to the future: The future is inherently uncertain.
- It involves cause and effect: a good definition of a specific risk identifies a situation such as 'inexperience staff' and a particular type of outcome, such as lower productivity.

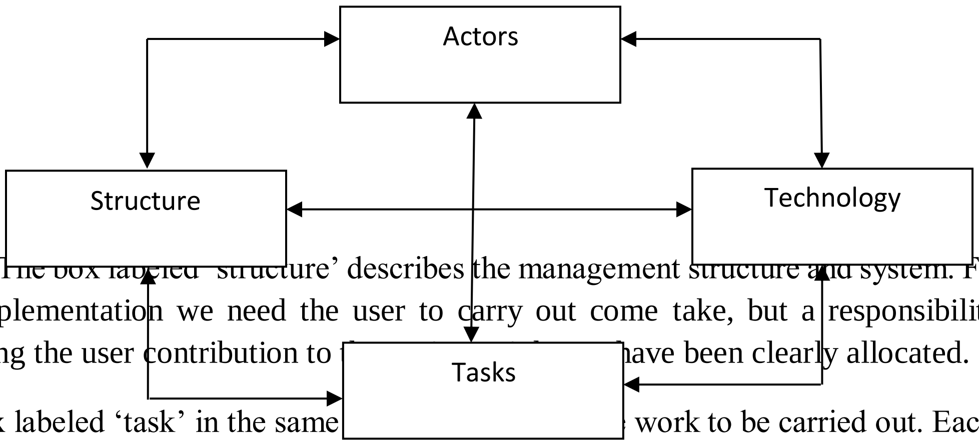
The different categories of risk:

Project risk is those that could prevent the achievement of the objectives given to the project managers and the project team.

Risk has been categorized in other ways. Kalle Lyytinen and his colleagues for instance, have proposed a socio technical model of risk, a diagrammatic representation.

The box labeled 'actors' refers to all the people involved in the development of the application in question. The typical risk in this area is that high staff turnover leads to information of value to the project being lost. For eg. If a software developer build a software component and then leaves before it has been fully tested, the team member taking over that component might find that their lack of familiarity with the software makes diagnosis and correction of faults difficult.

The box labeled 'technology' encompasses both the technology used to implement the application and that embedded in the delivered product. Risk here could relate to the appropriateness of the technologies and to possible fault within them, especially if the novel.



The box labeled 'structure' describes the management structure and system. For eg. The implementation we need the user to carry out come take, but a responsibility for managing the user contribution to have been clearly allocated.

The box labeled 'task' in the same work to be carried out. Each box is linked to all the remaining boxes.

1. **Dealing the risk:**

Planning for risk includes these steps:

1. Risk identification
2. Risk analysis and prioritization
3. Risk planning
4. Risk monitoring

The two main approaches to the identification of risks are the use of the check list and brainstorming.

Checklists:

Checklists are simply list of the risk that has been found to occur regularly in software development project. Two check list-that of Lyytinen and his colleagues and the ISPL/Euro method model-have already been mentioned, but other exists including a specialized list of software development risk by Barry Boehm- a modified version which is appearing in the following table.

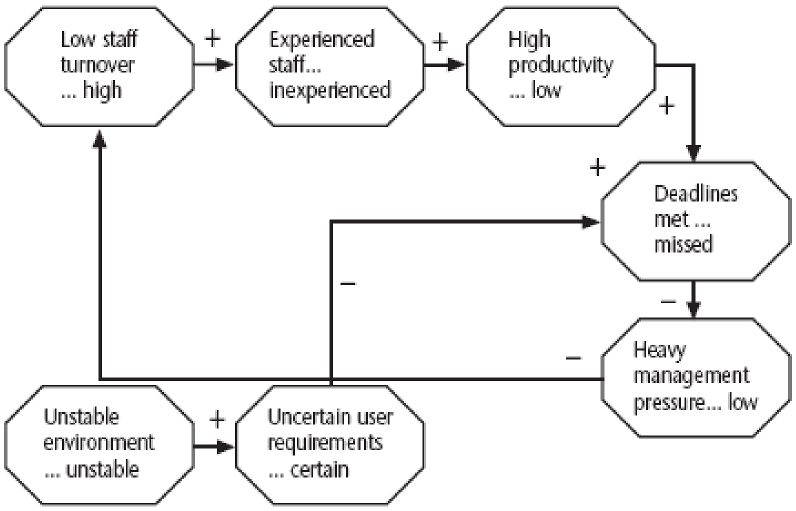
Risk	***Risk reduction techniques***
Personnel shortfalls	Staffing with top talent; job matching; teambuilding; training and career development; early scheduling of key personnel
Unrealistic time and cost estimates	Multiple estimation techniques; design to cost; incremental development; recording and analysis of past projects; standardization of methods

Developing the wrong software functions	Improved software evaluation; formal specification methods; user surveys; prototyping; early user manuals
Developing the wrong user interface	Prototyping; task analysis; user involvement
Personnel shortfalls	Staffing with top talent; job matching; teambuilding; training and career development; early scheduling of key personnel
Unrealistic time and cost estimates	Multiple estimation techniques; design to cost; incremental development; recording and analysis of past projects; standardization of methods
Developing the wrong software functions	Improved software evaluation; formal specification methods; user surveys; prototyping; early user manuals
Developing the wrong user interface	Prototyping; task analysis; user involvement

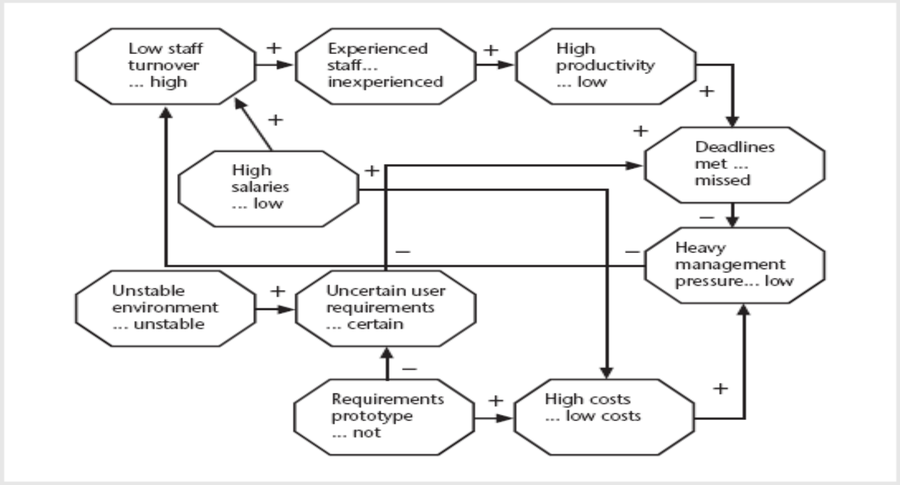
Brainstorming:

Representatives of the main stakeholders can e bought together, ideally, once some kind of preliminary plan has been drafted then identifies using their individual knowledge of different parts of the project, the particular problem that might occur. Brainstorming can also been used to identify the possible solution to the problems that emerge. One useful outcome of such an approach is that we collaborative approaches may generate the sense of owner ship in the project. The process that is beneficial explicitly asked stakeholders about their anxieties and then explores way of reducing those concerns.

Casual mapping:


Low staff turnover ... high
+
Experienced staff... inexperienced
+
High productivity ... low
+
+
Deadlines met ... missed
–
–
Heavy management pressure... low
–
Unstable environment ... unstable
+
Uncertain user requirements ... certain

- The idea here is to get the major stakeholders together and to brainstorm collectively the things that could go wrong. The causes of the problems identified are traced back using the mapping technique which identifies the project factors (or ‘concept variables’) that people see as being important and the causal links between them. These links can be positive or negative.
- Where possible, for each factor, positive and negative aspects are identified e.g. ‘stable…unstable requirements’.


Low staff turnover ... high
Experienced staff... inexperienced
High productivity ... low
Deadlines met ... missed
High salaries ... low
Heavy management pressure... low
Unstable environment ... unstable
Uncertain user requirements ... certain
Requirements prototype ... not
High costs ... low costs
+
−

- Figure 7.4 A causal/cognitive map of a problem area with proposed solutions

- Once a causal map has been drawn up identifying possible negative outcomes and their causes, the map can be modified to introduce policies or interventions which should reduce or mitigate the effects of the negative outcomes.
- Often a risk reduction activity can actually introduce new risks. The use of consultants to offset the effects of skill shortages is an example of this. Causal mapping can help identify such adverse side-effects.

Risk assessment:

- The common problem with risk identification, particularly for the more anxious, is that a list of risk is potentially endless. Some way is therefore needed of distinguishing the more damaging and likely risks. This can be done by estimating the risk exposure for each risk using the formula:
- Risk exposure (RE) = (potential damage) x (probability of occurrence)

- If there were 100 people chipping in $5,000 each, there would be enough for the 1 in 100 chance of the flooding. If there were 2 floods then the system collapses!

Risk planning:

Risks can be dealt with by:

- Risk acceptance – the cost of avoiding the risk may be greater than the actual cost of the damage that might be inflicted
- Risk avoidance – avoid the environment in which the risk occurs e.g. buying an OTS application would avoid a lot of the risks associated with software development e.g. poor estimates of effort.
- Risk reduction – the risk is accepted but actions are taken to reduce its likelihood e.g. prototypes ought to reduce the risk of incorrect requirements
- Risk transfer – the risk is transferred to another person or organization. The risk of incorrect development estimates can be transferred by negotiating a fixed price contract with an outside software supplier.
- Risk mitigation – tries to reduce the impact if the risk does occur e.g. taking backups to allow rapid recovery in the case of data corruption

Evaluate the risks with PERT technique

1. Applying the PERT Technique:

The method is very similar to the CPM technique but instead of using a single estimate for the duration of each tack, pert requires three estimates.

- *Most likely time (m)* the time we would expect the task to take normally
- *Optimistic time (a)* the shortest time could be realistically be expected
- *Pessimistic (b)* worst possible time (only 1% chance of being worse, say)

Some straightforward activities might have little uncertainty and therefore have a low standard deviation, while others have more uncertainty and would have a bigger standard deviation.

Pert then combines these three estimates to form a single expected duration using the formula:

'expected time' $t_e = (a + 4m + b) / 6$

A quantitative measure of the degree of uncertainity of activity duration estimate may be obtained by calculating the standard deviation S of an activity time using the formula:

Activity	**Optimistic**	**Most likely**	**Pessimistic**	**Expected**	**Standard deviation**
	(a)	(m)	(b)	(te)	(s)
A	5	6	8	6.17	0.50
B	3	4	5	4.00	0.33
C	2	3	3	2.83	0.17
D	3.5	4	5	4.08	0.25
E	1	3	4	2.83	0.50
F	8	10	15	10.50	1.17
G	2	3	4	3.00	0.33
H	2	2	2.5	2.08	0.08

‘activity standard deviation’ S = (b-a)/6

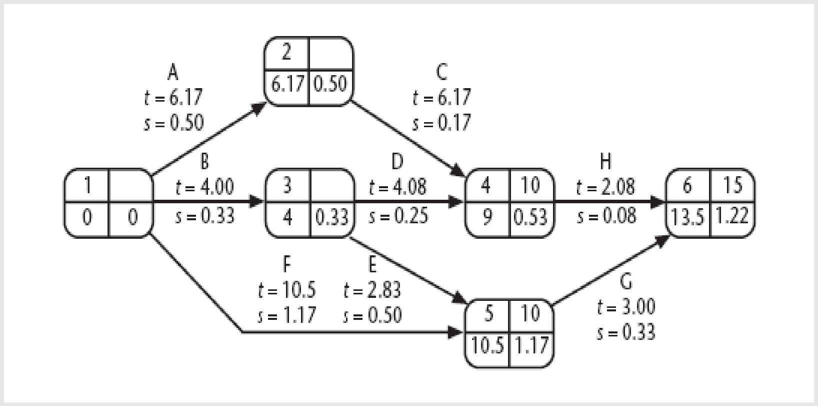


Figure 7.9 The PERT network with three target dates and calculated event standard deviations

The PERT technique uses the following three step method for calculating the probability of meeting or missing a target date:

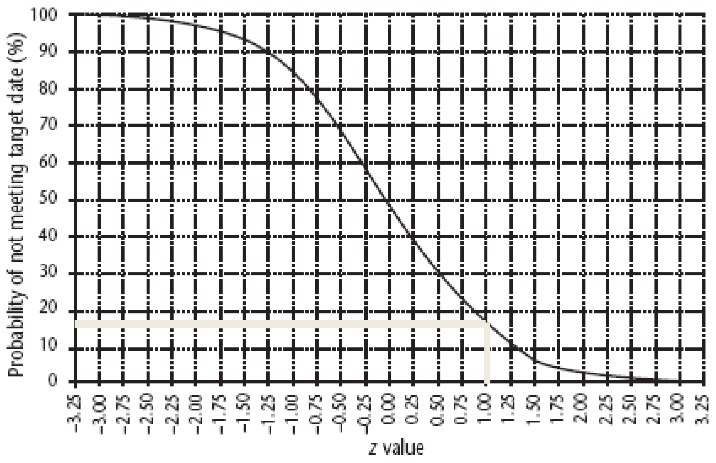
- Calculate the standard deviation of each project event;
- Calculate the z value for each event that has a target date;
- Convert z values to a probabilities.

Calculate the z value thus

$$\mathbf{z = (T - t_e)/s}$$

Where te is the expected date and the T the target date.

The z value for event 4 is (10-9.00)/0.53=1.8867.



There is about a 17% chance of not meeting the target of 52 days.