UNIT – II

Software Project Management Renaissance

Conventional Software Management, Evolution of Software Economics, Improving Software Economics, The old way and the new way.

Life-Cycle Phases and Processartifacts

Engineering and Production stages, inception phase, elaboration phase, construction phase, transition phase, artifact sets, management artifacts, engineering artifacts and pragmatic artifacts, model based software architectures

Part –I Software Project Management Renaissance

Conventional Software Management

1. The best thing about software is its flexibility:
   - It can be programmed to do almost anything.
2. The worst thing about software is its flexibility:
   - The "almost anything" characteristic has made it difficult to plan, monitor, and control software

development.

3. In the mid-1990s, three important analyses were performed on the software engineering industry.

All three analyses given the same general conclusion:-

"The success rate for software

projects is very low". They Summarized as follows:

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

Software management process framework:

WATERFALLL MODEL

1. It is the baseline process for most conventional software projects have used.
2. We can examine this model in two ways:

IN THEORY

IN PRACTICE

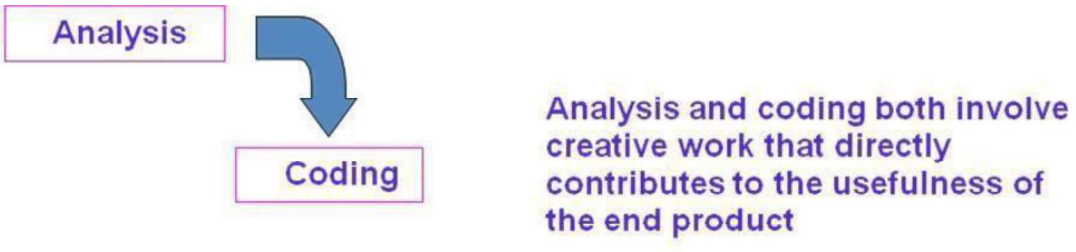
IN THEORY:-

In 1970, Winston Royce presented a paper called "Managing the Development of Large Scale Software Systems" at IEEE WESCON.

Where he made three primary points:

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

- coding

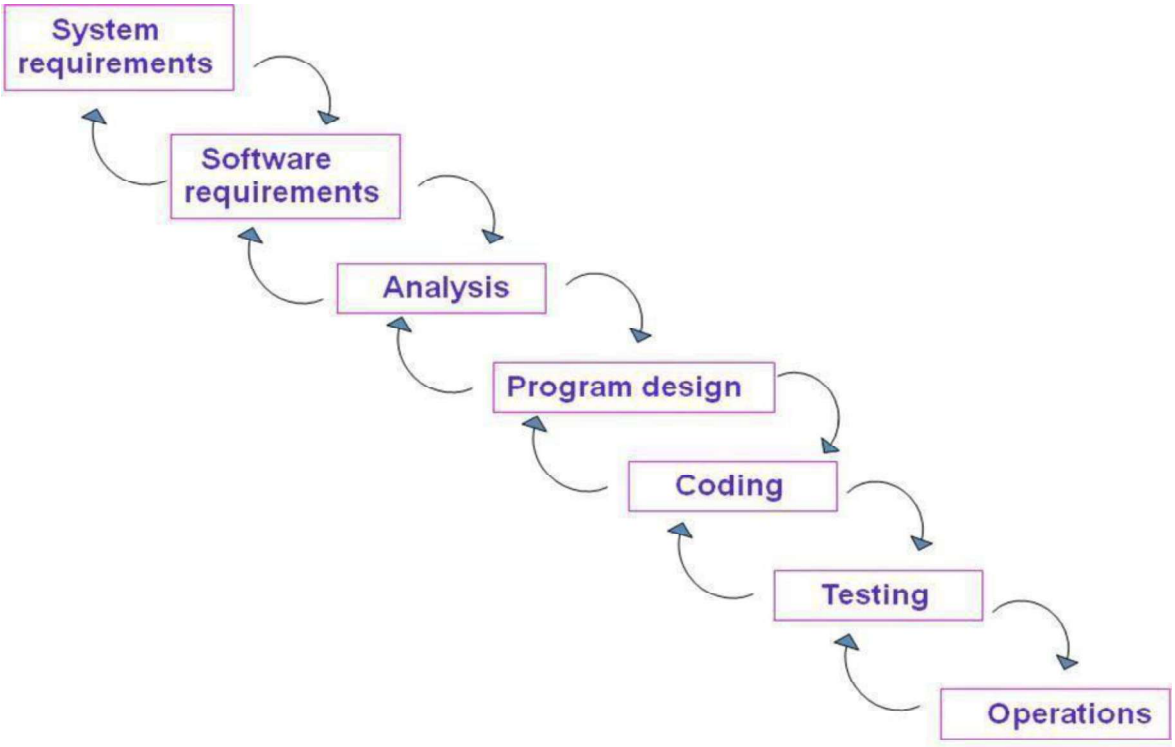


Waterfall Model Part 1: The two basic steps to build a program

2. In order to mange and control all of the intellectual freedom associated with software development one should follow the following steps:

- System requirements definition
- Software requirements definition
- Program design and testing

These steps addition to the analysis and coding steps



Waterfall Model Part 2: The large – scale system approach

3. Since the testing phase is at the end of the development cycle in the waterfall model, it may be risky and invites failure.

So we need to do either the requirements must be modified or a substantial design changes is warranted by breaking the software in to different pieces.

-**There are five improvements to the basic waterfall model that would eliminate most of the development risks are as follows:**

a) Complete program design before analysis and coding begin (program design comes first):-

- By this technique, the program designer give surety that the software will not fail because of storage, timing, and data fluctuations.
- Begin the design process with program designer, not the analyst or programmers.
- Write an overview document that is understandable, informative, and current so that every worker on the project can gain an elemental understanding of the system.

b) Maintain current and complete documentation (Document the design):-

-It is necessary to provide a lot of documentation on most software programs.

- Due to this, helps to support later modifications by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

c) Do the job twice, if possible (Do it twice):-

- If a computer program is developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations are concerned.
- "Do it N times" approach is the principle of modern-day iterative development.

d) Plan, control, and monitor testing:-

- The biggest user of project resources is the test phase. This is the phase of greatest risk in terms of cost and schedule.
- In order to carryout proper testing the following things to be done:

i) Employ a team of test specialists who were not responsible for the original design.

ii) Employ visual inspections to spot the obvious errors like dropped minus signs, missing factorsof two, jumps to wrong addresses.

iii) Test every logic phase.

iv) Employ the final checkout on the target computer.

e) Involve the customer:-

- It is important to involve the customer in a formal way so that he has committed himself at earlier points before final delivery by conducting some reviews such as,

i) Preliminary software review during preliminary program design step.

ii) Critical software review during program design.

iii) Final software acceptance review following testing.

IN PRACTICE:-

- Whatever the advices that are given by the software developers and the theory behind the waterfall model, some software projects still practice the conventional software management approach.

Projects intended for trouble frequently exhibit the following symptoms:

i) Protracted (delayed) integration

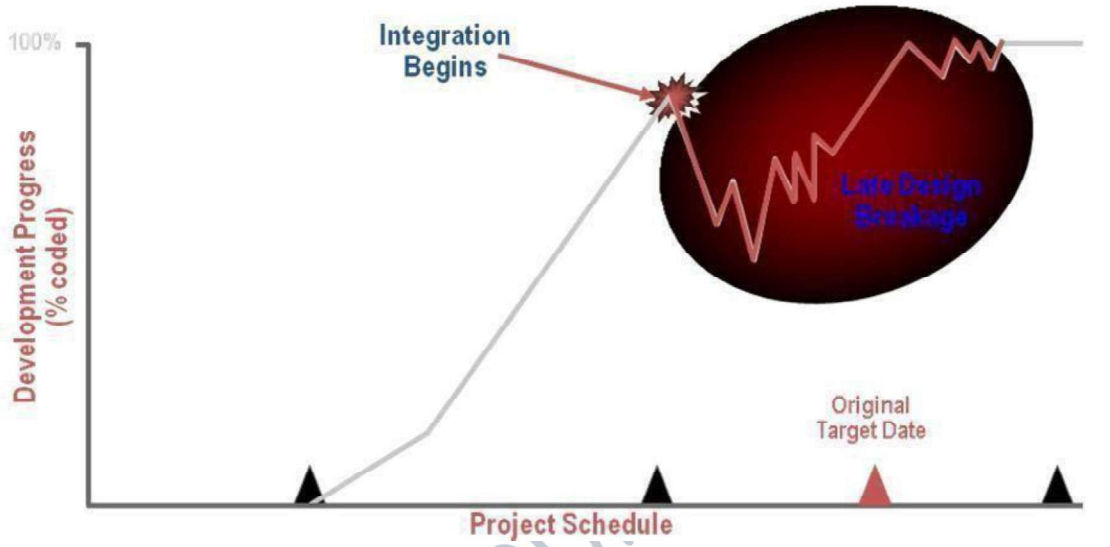
- In the conventional model, the entire system was designed on paper, then implemented all at once, then integrated. Only at the end of this process was it possible to perform system testing to verify that the

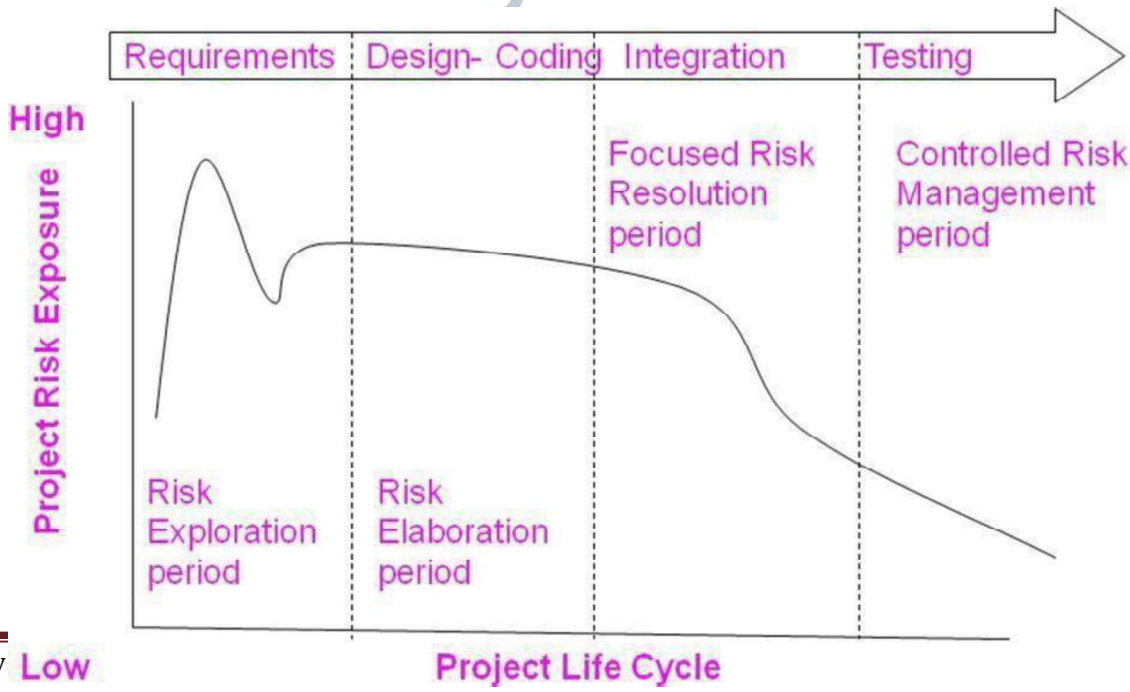
fundamental architecture was sound.

What Happens in Practice

Sequential activities:

Requirements → Design → Code → Integration → Test





ii) Late Risk Resolution

- A serious issues associated with the waterfall life cycle was the lack of early risk resolution.

The risk profile of a waterfall model is,

- It includes four distinct periods of risk exposure, where risk is defined as "the probability of missing a cost, schedule, feature, or quality goal".

iii) Requirements-Driven Functional Decomposition

-Traditionally, the software development process has been requirement-driven: An attempt is made to provide a precise requirements definition and then to implement exactly those requirements.

-This approach depends on specifying requirements completely and clearly before other development activities begin.

- It frankly treats all requirements as equally important.
- Specification of requirements is a difficult and important part of the software development process. **iv) Adversarial Stakeholder Relationships**

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

- The contractor prepared a draft contact-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.
- The customer was expected to provide comments (within 15 to 30 days)
- The contractor integrated these comments and submitted a final version for approval (within 15 to 30 days)

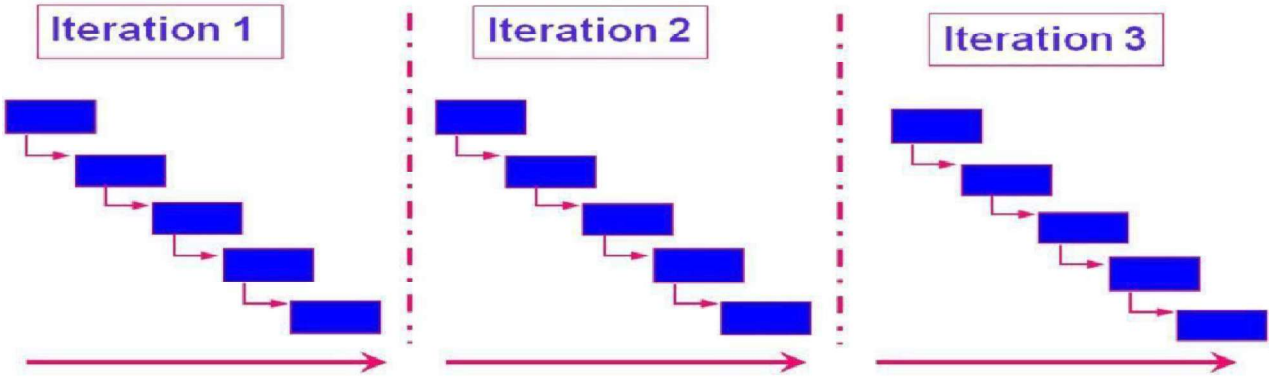
Project Stakeholders :

- Stakeholders are the people involved in or affected by project activities
- Stakeholders include
  - the project sponsor and project team
  - support staff
  - customers
  - users
  - suppliers
  - opponents to the project

v) Focus on Documents and Review Meetings

- The conventional process focused on various documents that attempted to describe the software product.
- Contractors produce literally tons of paper to meet milestones and demonstrate progress to stakeholders, rather than spend their energy on tasks that would reduce risk and produce quality software.
- Most design reviews resulted in low engineering and high cost in terms of the effort and schedule involved in their preparation and conduct.

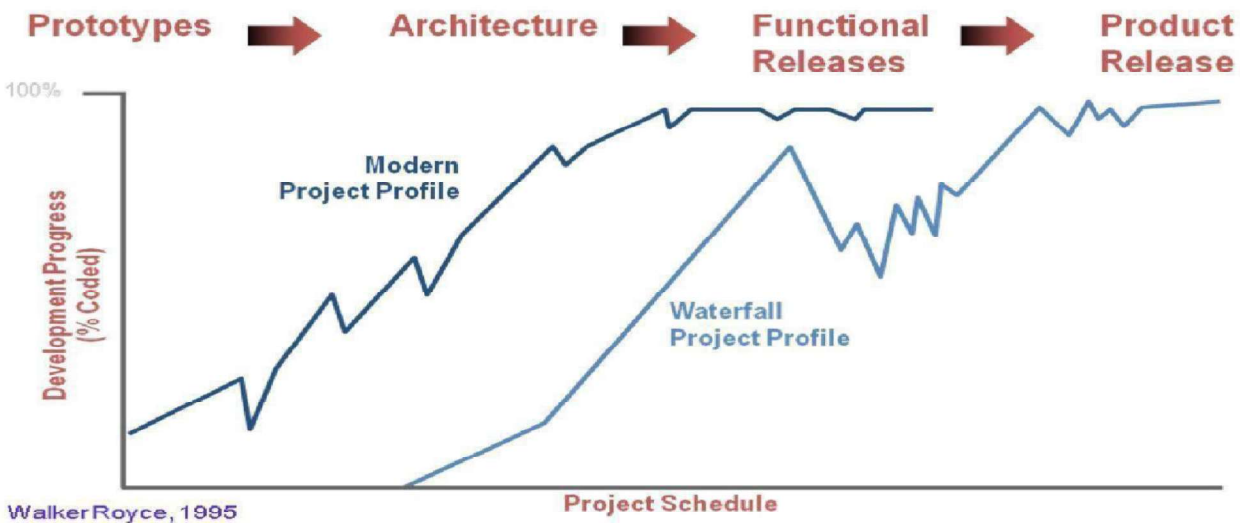
Iterative Development



- Earliest iterations address greatest risks
- Each iteration produces an executable release
- Each iteration includes integration and test

Better Progress Profile

Sequential phases, but iterative activities



Iterative Development Phases

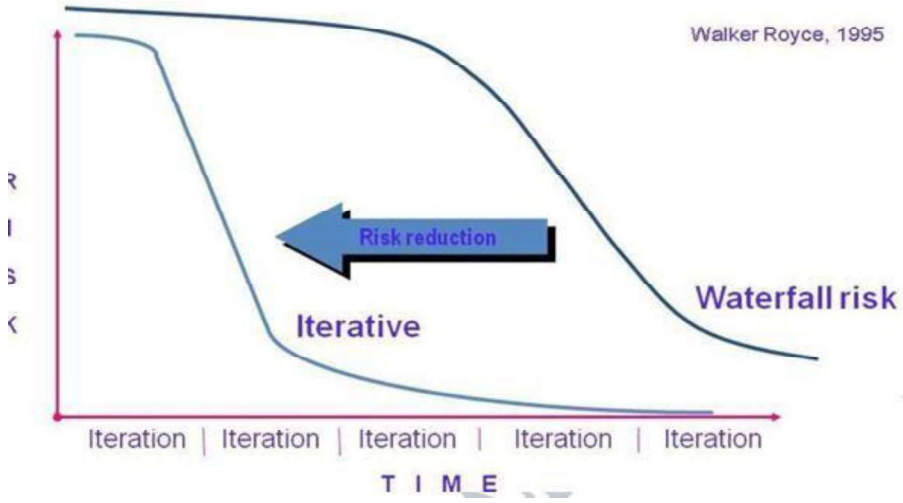
Inception: Agreement on overall scope
- Vision, high-level requirements, business case
- Not detailed requirements

Elaboration: Agreement on design approach
- Baseline architecture, most requirements detailed
- Not detailed design

Construction: Apply approach
- Working product, system test complete

Transition: Validate solution
- Stakeholder acceptance

Accelerate Risk Reduction



Barry Boehm"s Top 10 "Industrial Software Metrics":

1) Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.
2) You can compress software development schedules 25% of nominal (small), but no more.
3) For every $1 you spend on development, you will spend $2 on maintenance.
4) Software development and maintenance costs are primarily a function of the number of source lines of code.
5) Variations among people account for the biggest difference in software productivity.
6) The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.
7) Only about 15% of software development effort is devoted to programming.
8) Software systems and products typically cost 3 times as much per SLOC as individual software programs. Software-system products cost 9 times as much.
9) Walkthroughs catch 60% of the errors.
10) 80% of the contribution comes from 20% of the contributors.

    - 80% of the engineering is consumed by 20% of the requirements.
    - 80% of the software cost is consumed by 20% of the components.
    - 80% of the errors are caused by 20% of the components.
    - 80% of the software scrap and rework is caused by 20% of the errors.
    - 80% of the resources are consumed by 20% of the components.
    - 80% of the engineering is accomplished by 20% of the tools.
    - 80% of the progress is made by 20% of the people.

Evolution of Software Economics

Project Sizes :

- Size as team strength could be :
  - Trivial (Minor) Size: 1 person
  - Small Size: 5 people
  - Moderate Size: 25 people
  - Large Size: 125 people
  - Huge Size: 625 people
- The more the size, the greater are the costs of management overhead, communication, synchronizations among various projects or modules, etc.

Reduce Software Size:

The less software we write, the better it is for project management and for product quality

- The cost of software is not just in the cost of „coding" alone; it is also in Analysis of requirements
  - Design
  - Review of requirements, design and code
  - Test Planning and preparation
  - Testing
  - Bug fix
  - Regression testing
  - „Coding" takes around 15% of development cost
- Clearly, if we reduce 15 hrs of coding, we can directly reduce 100 hrs of development effort, and also reduce the project team size appropriately !

Size reduction is defined in terms of human-generated source code.
Most often, this might still mean that the computer-generated executable code is at least the same or even more

- Software Size could be reduced by
  - Software Re-use
  - Use of COTS (Commercial Off-The Shelf Software)
  - Programming Languages

PRAGMATIC SOFTWARE ESTIMATION:

- If there is no proper well-documented case studies then it is difficult to estimate the cost of the

software. It is one of the critical problem in software cost estimation.

- But the cost model vendors claim that their tools are well suitable for estimating
- iterative development projects.
- In order to estimate the cost of a project the following three topics should be considered,

1) Which cost estimation model to use.

2) Whether to measure software size in SLOC or function point.

3) What constitutes a good estimate.

- There are a lot of software cost estimation models are available such as, COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20.
- Of which COCOMO is one of the most open and well-documented cost estimation models
- The software size can be measured by using

1) SLOC 2) Function points

- Most software experts argued that the SLOC is a poor measure of size. But it has some value in the software Industry.
- SLOC worked well in applications that were custom built why because of easy to automate and instrument.
- Now a days there are so many automatic source code generators are available and there are so many advanced higher-level languages are available. So SLOC is a uncertain measure.
- The main advantage of function points is that this method is independent of the technology and is therefore a much better primitive unit for comparisons among projects and organizations.
- The main disadvantage of function points is that the primitive definitions are abstract and measurements are not easily derived directly from the evolving artifacts.
- Function points is more accurate estimator in the early phases of a project life cycle. In later phases, SLOC becomes a more useful and precise measurement basis of various metrics perspectives.
- The most real-world use of cost models is bottom-up rather than top-down.
- The software project manager defines the target cost of the software, then manipulates the parameters and sizing until the target cost can be justified.

Improving Software Economics

- It is not that much easy to improve the software economics but also difficult to measure and validate.
- There are many aspects are there in order to improve the software economics they are, Size, Process, Personnel, Environment and quality.

- These parameters (aspects) are not independent they are dependent. For example, tools enable size reduction and process improvements, size- reduction approaches lead to process changes, and process improvements drive tool requirements.
- GUI technology is a good example of tools enabling a new and different process. GUI builder tools permitted engineering teams to construct an executable user interface faster and less cost.
- Two decades ago, teams developing a user interface would spend extensive time analyzing factors, screen layout, and screen dynamics. All this would done on paper. Where as by using GUI, the paper descriptions are not necessary.

Along with these five basic parameters another important factor that has influenced software technology improvements across the board is the ever- increasing advances in hardware Performance.

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

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

REDUCING SOFTWARE PRODUCT SIZE:

- By choosing the type of the language
- By using Object-Oriented methods and visual modeling
- By reusing the existing components and building reusable components &

By using commercial components,we can reduce the product size of a software.

OBJECT ORIENTED METHODS AND VISUAL MODELING:

- There has been a widespread movements in the 1990s toward Object- Oriented technology.
- Some studies concluded that Object-Oriented programming languages appear to benefit both software productivity and software quality. One of such Object-Oriented method is UML-Unified Modeling Language.

Booch described the following three reasons for the success of the projects that are using Object-Oriented concepts:

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

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

3) An OO-architecture provides a clear separation among different elements of a system, crating firewalls that prevent a change in one part of the system from the entire architecture.

He also suggested five characteristics of a successful OO-Project,

1) A cruel focus on the development of a system that provides a well understood collection of essential minimal characteristics.

2) The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail.

3) The effective use of OO-modeling.

4) The existence of a strong architectural vision.

5) The application of a well-managed iterative and incremental development life cycle.

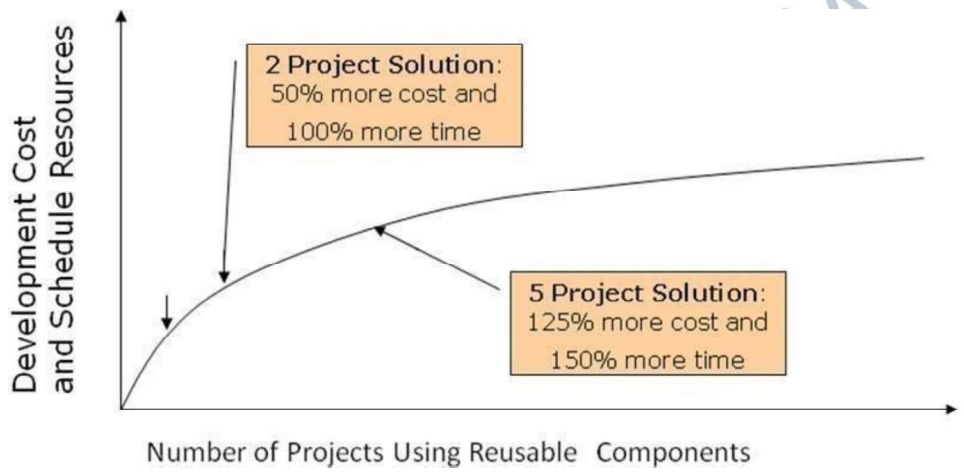
REUSE:

Organizations that translates reusable components into commercial products has 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.



COMMERCIAL COMPONENTS

It is an Organization‟s policies, procedures, and practices for pursuing a software-intensive line of business.

The focus of this process is of organizational economics, long-term strategies, and a software ROI.

- **Macro process:**

A project‟s policies, and practices for producing a complete software product within certain cost, schedule, and quality constraints.

The focus of the macroprocess is on creating an sufficient instance of the metaprocess for a specific set of constraints.

- **Micro process:**

A projects team‟s policies, procedures, and practices for achieving an artifact of a software process.

The focus of the microprocess is on achieving an intermediate product baseline with sufficient functionality as economically and rapidly as practical.

The objective of process improvement is to maximize the allocation of resources to productive activities and minimize the impact of overhead activities on resources such as personnel, computers, and schedule.

Schedule improvement has at least three dimensions.

1. We could take an N-step process and improve the efficiency of each step.
2. We could take an N-step process and eliminate some steps so that it is now only an M-step process.
3. We could take an N-step process and use more concurrency in the activities being performed or the resources being applied.

IMPROVING TEAM EFFECTIVENESS:

- COCOMO model suggests that the combined effects of personnel skill and experience can have an impact on productivity as much as a factor of four over the unskilled personnel.

- Balance and coverage are two of the most important features of excellent teams. Whenever a team is in out of balance then it is vulnerable.

- It is the responsibility of the project manager to keep track of his teams. Since teamwork is much more important than the sum of the individuals.

Boehm – staffing principles:

- **The principle of top talent:** Use better and fewer people.
- **The principle of job matching:** Fit the tasks to the skills and 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.

4) The principle of team balance: Select people who will complement and synchronize with one another.

5) The principle of phase-out: Keeping a misfit on the team doesn"t benefit anyone.

In general, staffing is achieved by these common methods:

- If people are already available with required skill set, just take them
- If people are already available but do not have the required skills, re-train them
- If people are not available, recruit trained people
- If you are not able to recruit skilled people, recruit and train people

Staffing of key personnel is very important:

- Project Manager
- Software Architect

Important Project Manager Skills:

- **Hiring skills.** Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.
- **Customer-interface skill.** Avoiding adversarial relationships among stake- holders is a prerequisite for success.
- **Decision-making skill.** The jillion books written about management have failed to provide a clear definition of this attribute. We all know a good leader when we run into one, and decision-making skill seems obvious despite its intangible definition.
- **Team-building skill.** Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.
- **Selling skill.** Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives. In practice, selling requires continuous negotiation, compromise, and empathy.

Important Software Architect Skills:

- Technical Skills: the most important skills for an architect. These must include skills in both, the problem domain and the solution domain
- People Management Skills: must ensure that all people understand and implement the architecture in exactly the way he has conceptualized it. This calls for a lot of people management skills and patience.
- Role Model: must be a role model for the software engineers – they would emulate all good (and also all bad !) things that the architect does

IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

The following are the some of the configuration management environments which provide the foundation for executing and implementing the process:

Planning tools, Quality assurance and analysis tools, Test tools, and User interfaces provide crucial automation support for evolving the software engineering artifacts.

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

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

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

PEER INSPECTIONS: A PRAGMATIC VIEW:

- Analysis, prototyping, or experimentation
- Constructing design models
- Committing the current state of the design model to an executable implementation
- Demonstrating the current implementation strengths and weaknesses in the context of critical subsets of the use cases and scenarios
- Incorporating lessons learned back into the models, use cases, implementations, and plans

THE OLD WAY AND THE NEW

- Over the past two decades software development is a re-engineering process. Now it is replaced by advanced software engineering technologies.

- This transition is was motivated by the unsatisfactory demand for the software and reduced cost.

THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

Based on many years of software development experience, the software industry proposed so many principles (nearly 201 by – Davis‟s). Of which Davis‟s top 30 principles are:

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

2) High-quality software is possible: In order to improve the quality of the product we need to involving the customer, select the prototyping, simplifying design, conducting inspections, and hiring the best people.

3) Give products to customers early: No matter how hard you try to learn user‟s needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play withit.

4) Determine the problem before writing the requirements: Whenever a problem is raised most engineers provide a solution. Before we try to solve a problem, be sure to explore all the alternatives and don‟t be blinded by the understandable solution.

5)Evaluate design alternatives: After the requirements are greed upon, we must examine a variety of architectures and algorithms and choose the one which is not used earlier.

6)Use different languages for different phases: Our industry‟s main aim is to provide simple solutions to complex problems. In order to accomplish this goal choose different languages for different modules/phases ifrequired.

7)Minimize intellectual distance: We have to design the structure of a software is as close as possible to the real-world structure.

8)Put techniques before tools: An un disciplined software engineer with a tool becomes a dangerous, undisciplined software engineer.

9)Get it right before you make it faster: It is very easy to make a working program run faster than it is to make a fast program work. Don‟t worry about optimization during initial coding.

10)Inspect the code: Examine the detailed design and code is a much better way to find the errors than testing.

11) Good management is more important than good technology

12)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.

13)Follow with care: Everybody is doing something but does not make it right for you. It may be right, but you must carefully assess its applicability to your environment.

14)Take responsibility: When a bridge collapses we ask "what did the engineer do wrong?". Similarly if the software fails, we ask the same. So the fact is in every engineering discipline, the best methods can be used to produce poor results and the most out of date methods to produce stylish design.

15) 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.

16)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 firsttime.

17) Design for change. The architectures, components, and specification techniques you use must accommodate change.

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

vi) Use tools, but be realistic. Software tools make their users more efficient.

vii) **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.**

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

ix) **Use coupling and cohesion**. Coupling and cohesion are the best ways to measure software‟s inherent maintainability and adaptability.

x) **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.

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

xii) **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.**

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

THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT

1) Base the process on an architecture-first approach: (Central design element)

- Design and integration first, then production and test

2) Establish an iterative life-cycle process: (The risk management element)

- Risk control through ever-increasing function, performance, quality.

With today‟s sophisticated systems, it is not possible to define the entire problem, design the entire solution, build the software, then test the end product in sequence. Instead, and iterative process that refines the problem understanding, an effective solution, and an effective plan over several iterations encourages balanced treatment of all stakeholder objectives.

Major risks must be addressed early to increase predictability and avoid expensive downstream scrap and rework.

3) Transition design methods to emphasize component-based development: (The technology element)

Moving from LOC mentally to component-based mentally is necessary to reduce the

amount of human-generated source code and custom development.A component is a cohesive set of preexisting lines of code, either in source or executable format, with a defined interface and behavior.

Establish a change management environment: (The control element)

- Metrics, trends, process instrumentation

The dynamics of iterative development, include concurrent workflows by different teams working on shared artifacts, necessitates objectively controlled baseline.

4) Enhance change freedom through tools that support round-trip engineering: (The automation element)

- Complementary tools, integrated environment

Round-trip engineering is the environment support necessary to automate and synchronize engineering information in different formats. Change freedom is necessary in an iterative process.

5) Capture design artifacts in rigorous, model-based notation:

- A model-based approach supports the evolution of semantically rich graphical and textual design notations.

- Visual modeling with rigorous notations and formal machine- process able language provides more objective measures than the traditional approach of human review and inspection of ad hoc design representations in paper doc.

6) Instrument the process for objective quality control and progress assessment:

- Life-cycle assessment of the progress and quality of all intermediate product must be integrated into the process.

- The best assessment mechanisms are well-defined measures derived directly from the evolving engineering artifacts and integrated into all activities and teams.

7) Use a demonstration-based approach to assess intermediate artifacts:

Transitioning from whether the artifact is an early prototype, a baseline architecture, or a beta capability into an executable demonstration of relevant provides more tangible understanding of the design tradeoffs, early integration and earlier elimination of architectural defects.

8) Plan intermediate releases in groups of usage scenarios with evolving levels of detail:

9) Establish a configurable process that economically scalable:

No single process is suitable for all software developments. The process must ensure that there is economy of scale and ROI.

Architecture-first approach → The central design element

Design and integration first, then production and test

Iterative life-cycle process → The risk management element

Risk control through ever-increasing function, performance, quality

Component-based development → The technology element

Object-oriented methods, rigorous notations, visual modeling

Change management environment → The control element

Metrics, trends, process instrumentation

Round-trip engineering → The automation element

Complementary tools, integrated environments

"Software Project Management"
Walker Royce

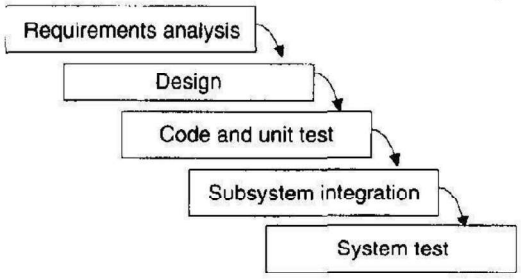
31/112

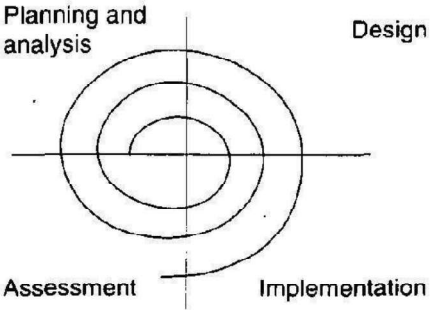
Waterfall Process

Requirements first
Custom development
Change avoidance
Ad hoc tools

Iterative Process

Architecture first
Component-based development
Change management
Round-trip engineering





PART-II Life cycle phases and process artifacts

LIFE-CYCLE PHASES

- If there is a well defined separation between "research and development" activities and"production" activities then the software is said to be in successful development process.
- Most of the software's fail due to the following characteristics ,

1) An overemphasis on research and development.
2) An overemphasis on production.

ENGINEERING AND PRODUCTION STAGES :

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

There are two stages in the software development process

1) **The engineering stage:** Less predictable but smaller teams doing design and production activities.

This stage is decomposed into two distinct phases *inception* and *elaboration.*

2) **The production stage:** More predictable but larger teams doing construction, test, and deployment activities. This stage is also decomposed into two distinct phases *construction* and *transition.*

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

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

1.

INCEPTION

PHASE:

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

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES:

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

2. ELABORATION PHASE

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

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES

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

3. CONSTRUCTION PHASE

During this phase all the remaining components and application features are developed software is integrated where ever required.

- If it is a big project then parallel construction increments are generated.

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES

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

4. TRANSITION PHASE

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

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

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES

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

Artifacts of the Process

- Conventional s/w projects focused on the sequential development of s/w artifacts:
- Build the requirements
- Construct a design model traceable to the requirements &
- Compile and test the implementation for deployment.

-This process can work for small-scale, purely custom developments in which the design representation, implementation representation and deployment representation are closely aligned.

- This approach is doesn't work for most of today's s/w systems why because of having complexity and are composed of numerous components some are custom, some reused, some commercial products.

THE ARTIFACT SETS

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

- *Set* represents a complete aspect of the system where as *artifact* represents interrelated information that is developed and reviewed as a single entity.
- The artifacts of the process are organized into five sets:

1) Management 2) Requirements 3) Design

4) Implementation 5) Deployment

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

THE MANAGEMENT SET

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

- **project personnel:**

project manager, architects, developers, testers, marketers, administrators

- **stakeholders:** funding authority, user, s/w project manager, organization manager, regulatory agency & between project personnel and stakeholders artifacts

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

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

THE ENGINEERING SETS:

1) REQUIREMENT SET:

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

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

2) DESIGN SET:

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

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

1) Analysis of the internal consistency and quality of the design model.
2) Analysis of consistency with the requirements models.
3) Translation into implementation and deployment sets and notations to evaluate the consistency and completeness and semantic balance between information in the sets.

4) Analysis of changes between the current version of the design model and previous versions.
5) Subjective review of other dimensions of quality.

3) IMPLEMENTATION SET:

- The implementation set include source code that represents the tangible implementations of components and any executables necessary for stand-alone testing of components.
- Executables are the primitive parts that are needed to construct the end product, including custom components, APIs of commercial components.
- Implementation set artifacts can also be translated into a subset of the deployment set. Implementation sets are human-readable formats that are evaluated, assessed, and measured through a combination of

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

4) DEPLOYMENT SET:

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

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

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

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

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

ARTIFACTS EVOLUTION OVER THE LIFE CYCLE

- Each state of development represents a certain amount of precision in the final system description.
- Early in the lifecycle, precision is low and the representation is generally high.

Eventually, the precision
of representation is high and everything is specified in full detail.

- At any point in the lifecycle, the five sets will be in different states of completeness. However, they should be at compatible levels of detail and reasonably traceable to one another.
- Performing detailed traceability and consistency analyses early in the life cycle

i.e. when precision is low and changes are frequent usually has a low ROI. **Inception phase:** It mainly focuses on critical requirements, usually with a secondary focus on an initial deployment view, little implementation and high- level focus on the design architecture but not on design detail.

Elaboration phase: It include generation of an executable prototype, involves subsets of development in all four sets. A portion of all four sets must be evolved to some level of completion before an architecture baseline can be established.

Fig: Life-Cycle evolution of the artifact sets

Construction: Its main focus on design and implementation. In the early stages the main focus is on the depth of the design artifacts. Later, in construction, realizing the design in source code and individually tested

components. This stage should drive the requirements, design, and implementation sets almost to completion. Substantial work is also done on the deployment set, at least to test one or a few instances of the programmed system through alpha or beta releases.

Transition: The main focus is on achieving consistency and completeness of the deployment set in the context of another set. Residual defects are resolved, and feedback from alpha, beta, and system testing is incorporated.

MANAGEMENT ARTIFACTS:

 Development of WBS is dependent on product management style , organizational culture, custom performance, financial constraints and several project specific parameters.

• The WBS is the architecture of project plan. It encapsulate change and evolve with appropriatelevel of details.

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

• A WBS provides the following information structure

- A delineation of all significant tasks.
- A clear task decomposition for assignment of responsibilities.

- A framework for scheduling ,debugging and expenditure tracking.

-Most systems have first level decomposition subsystem. subsystems are then decomposed into their components

- Therefore WBS is a driving vehicle for budgeting and collecting cost.
- The structure of cost accountability is a serious project planning constraints.

Business case:

- Managing change is one of the fundamental primitives of an iterative development process.
- This flexibility increases the content, quality, and number of iterations that a project can achieve within a given schedule.
- Once software is placed in a controlled baseline, all changes must be formally tracked and managed.
- Most of the change management activities can be automated by automating data entry and maintaining change records online.

Engineering Artifacts

Engineering Artifacts are captured in rigorous engineering notations such as UML, programming languages, or executable machine codes. Three Engineering Artifacts are:

1. Vision Document.
2. Architecture Description.
3. S/W User Manual.

Vision Document

The source for capturing the expectations among stakeholders.

- Written from the users' perspective.
- Focus is on essential features of the system, and the acceptable levels of quality.
- Includes the operational concept

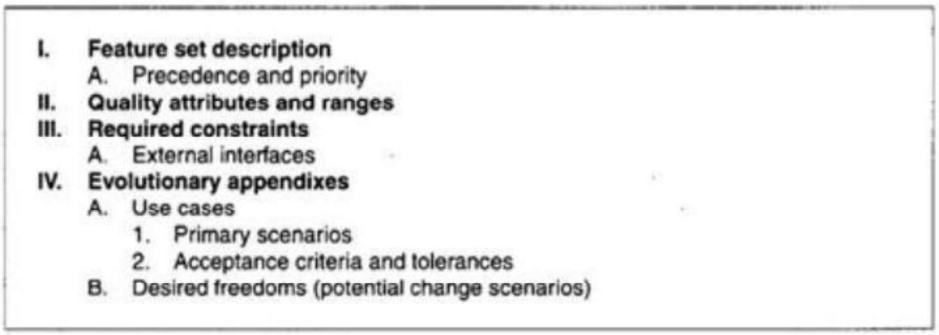


FIGURE 6-9. *Typical vision document outline*

Pragmatic Artifacts

- Pragmatic Meaning is dealing with things sensibly and realistically in a way that is based on practical rather than theoretical considerations.
- People want to review information but don't understand the language of the artifact.
- People want to review the information but don't have access to the tools.
- Human readable engineering artifacts should use rigorous notations that are complete, consistent and used in a self documenting manner.
- Useful documentation is self defining: It is documentation that gets used.
- Paper is tangible(perceptible by touch): electronic artifacts are too easy to change.

Model-Based Software Architectures

INTRODUCTION:

Software architecture is the central design problem of a complex software system in the same way an *architecture* is the software system design.

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

ARCHITECTURE : A MANAGEMENT PERSPECTIVE

- The most critical and technical product of a software project is its architecture
- If a software development team is to be successful, the interproject communications, as captured in software architecture, must be accurate and precise.

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

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

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

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

ARCHITECTURE: A TECHNICAL PERSPECTIVE

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

Most real-world systems require four types of views:

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

The design set include all UML design models describing the solution space.

- The design, process, and use case models provide for visualization of the logical and behavioral aspect of the design.
- The component model provides for visualization of the implementation set.

The deployment model provides for visualization of the deployment set.

1. The *use case view* describes how the system's critical use cases are realized by elements of the design model. It is modeled statistically by using use case diagrams, and dynamically by using any of the UML behavioral diagrams.
2. The *design view* describes the architecturally significant elements of the design model. It is modeled statistically by using class and object diagrams, and dynamically using any of the UML behavioral diagrams.
3. The *process view* addresses the run-time collaboration issues involved in executing the architecture on a distributed deployment model, including logical software topology, inter process communication, and state mgmt. it is modeled statistically using deployment diagrams, and dynamically using any of the UML behavioral diagrams.
4. The *component view* describes the architecturally significant elements of the implementation set. It is modeled statistically using component diagrams, and dynamically using any of the UML behavioral diagrams.

The *deployment view* addresses the executable realization of the system, including the allocation of logical processes in the distributed view to physical resources of the deployment network. It is modeled statistically using deployment diagrams, and dynamically using any of UML behavioral diagrams.

Architecture descriptions take on different forms and styles in different organizations and domains. At any given time, an architecture requires a subset of artifacts in engineering set.

- An architecture baseline is defined as a balanced subset of information across all sets, where as an architecture description is completely encapsulated within the design set.

Generally architecture base line include:

1) Requirements 2) Design

3) Implementation 4) Deployment