UNIT-I CONVENTIONAL SOFTWARE MANAGEMENT

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

→ The worst thing about the software is also its flexibility "The almost anything" characterstic has made it difficult to plan, monitor, and control software development.

→ In the mid-1990s, at least three important analyses of the state of the software Engineering industry were performed.

→ All three analyses reached the same general conclusion The sucess rate for software project is very low.

They can be summarized as follows:

* Software development is still highly unpredictable. only about 10% of software projects are delivered successfully within initial budget and schedule estimates.

* Management discipline is more of a discriminator in success or failure than are technology advances.

* The level of software scrap and rework is indicative of an immature process

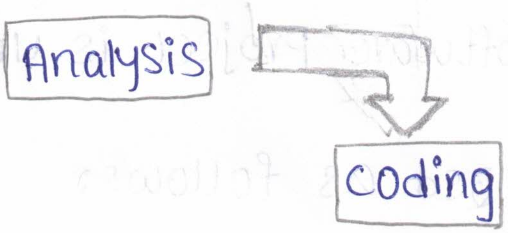
The Waterfall Model:

→ In Software Engineering the water fall model is conventional Software development Process and it treated as benchmark of that Process.

→ In 1970, my father, winston Royce, Presented a paper titled "Managing the Development of Large Scale software systems" at IEEE WESCON.

The waterfall model made three Primary Points.

* There are two Essential steps common to the development of a computer programs are analysis and coding.

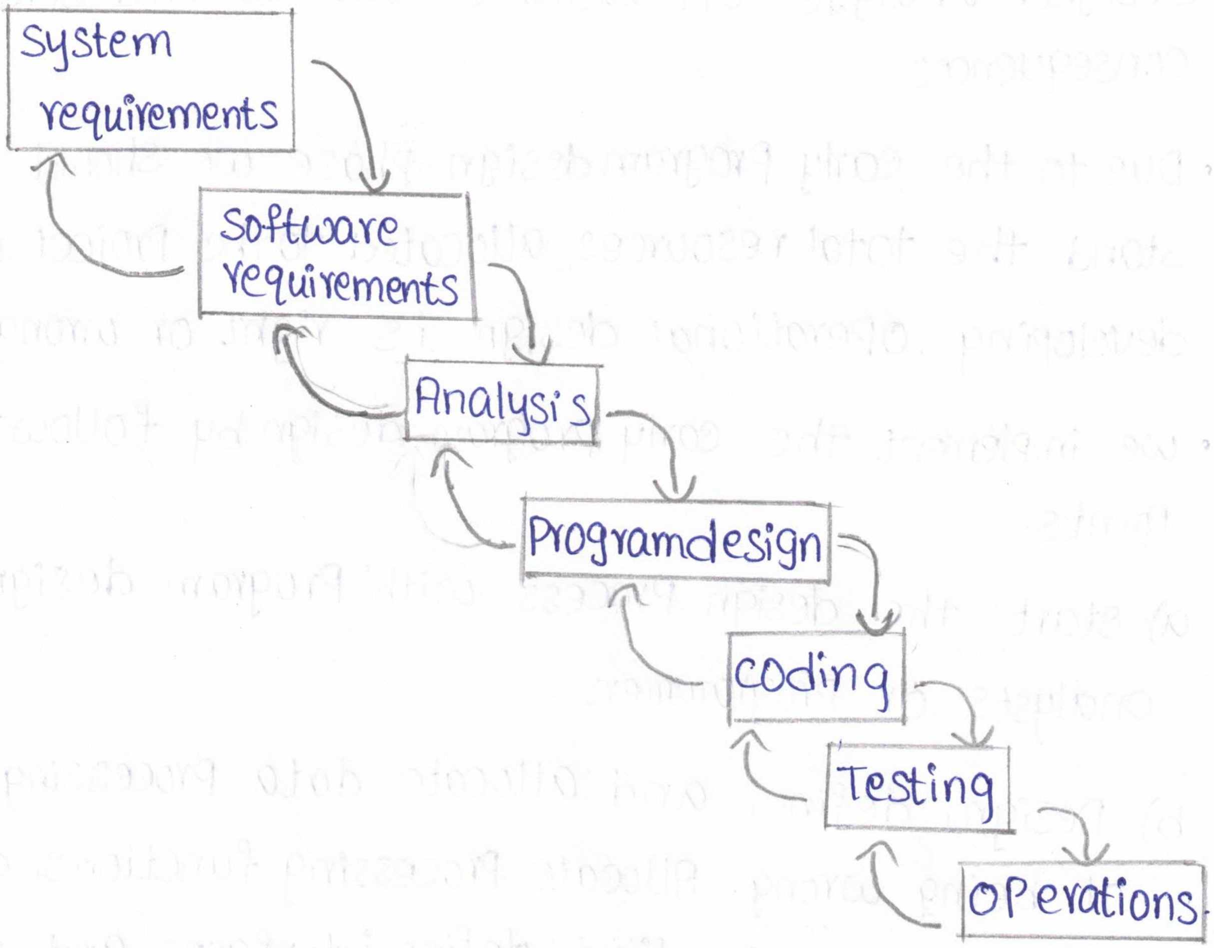


* To control intellectual freedom associated with software development, we introduce system requirements, software requirements, Program design and testing.

* The basic framework in the waterfall model is risky and invites failure.

The following diagram shows waterfall model Profile and the basic steps.

The large-scale system approach.



The five Necessary improvements for this approach to work

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

1. Program design comes first:- we should take-up Preliminary Program design as the first step even before the requirements and analysis Phases.

- At this Point the Program designer assures that software will

- when we performing analysis as the next Phase, the Program designer analyze Operational constraints and senses the consequences.

- Due to the early Program design Phase we should Easily understand the total resources allocated to the Project are insufficiet developing operational design is right or wrong or etc..

- we implement the early program design by follow the following Points.

a) start the design Process with Program designer not with analysts or programmers.

b) Design, define, and allocate data Processing modes even at being wrong. Allocate Processing functions, design DB. Allocate execution time, define interfaces and operating system, describe input and output Process, define Preliminary operating Procedures.

c) write an overview document that is understandable, informative, and current so that every worker on the Project can gain and elemental understanding of the system.

2) Document the design:- we know that document design is required a lot for software Programs.

- The Programmers, analysts, designers are mostly depending on documentation to finish their work.

a) The software designers communicate with interfacing.

b) During early Phases documentation is the design.

c) The value of documentation is to support later modifications by separate test team, seperate maintenance team and operational Personnel who are not software literate.

3) Do it twice:- The software delivered to the customer for operational Purpose is actually the second version, after conse-rding critical design and operational issues.

- In first version of software development, the team have a special broad competence where they can quickly sense trouble spots in the design, model them, model alternatives.

4) Plan, control, and monitor testing :- The Combination of Man Power, computer time and management is called test Phase. This Phase has greater risk in terms of cost and schedule. The following are the important things in the test Phase;

a) Employ a team of test specialists who were not responsible for the orginal design

b) Employ visual inspections to spot the obvious errors like dropped minus, missing factors of two, jumps to wrong address

c) Test every logic Path.

d) Employ the final checkout on the target computer.

5) Involve the customer :

Some times Software design going to wide interpretations even after Previous agreement. So it is important to involve the customer in a formal way so that he has commited himself at earlier Points before final delivery.

- In sight, judgment and commitment of a customer can boost the Project development Process. It may done at Preliminary Software review following the Preliminary Program design step, Sequence of final Software acceptance review.
- Most review meetings has low Engineering values and high cost in terms of effort and schedule involved in their Preparation and conduct.

Conventional Software Management Performance

Barry Bhoom describes the objective characterization of state of software development. Many of the metrics describe fundamental Economic relationships that resulted from conventional software Process Practiced over the long Period.

1. finding and fixing a software Problem after delivery cost 100 times more than finding and fixing the problem in Early design Phases.

2. You can compress software development schedules 25%

3. For every one rupee we spend on development, we will spend two rupees on maintenance.

4. Software development and maintenance costs are primarily a function of the number of source lines of code.

5. Variations among people account for the biggest differences in software productivity.

6. The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.

7. Only about 15% of software development effort is devoted to programming.

8. Software systems and products typically cost 3 times as much per SLOC as individual software programs. Software - system products cost 9 times as much.

9. Walk throughs catch 60% of the errors.

10. 80% of the contribution comes from 20% of the contributor

* 80% of the Engineering is consumed by 20% of requirements
* 80% of the software cost is consumed by 20% of component
* 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

Software Economics :-

The Most Software cost models can be abstracted into a function of five basic Parameters: Size, Process, Personnel environment, and required quality.

1. The size of end product, which is typically quantified in terms of the Number of source instructions or the Number of function Points required to develop the required functionality.

2. The Process used to Produce the end Product, in Particular the ability of the Process to avoid non-value-adding activities.

3. The capabilites of Software Engineering Personnel, and Particularly their experience with the computer science issues and the applications domain issues of the Project.

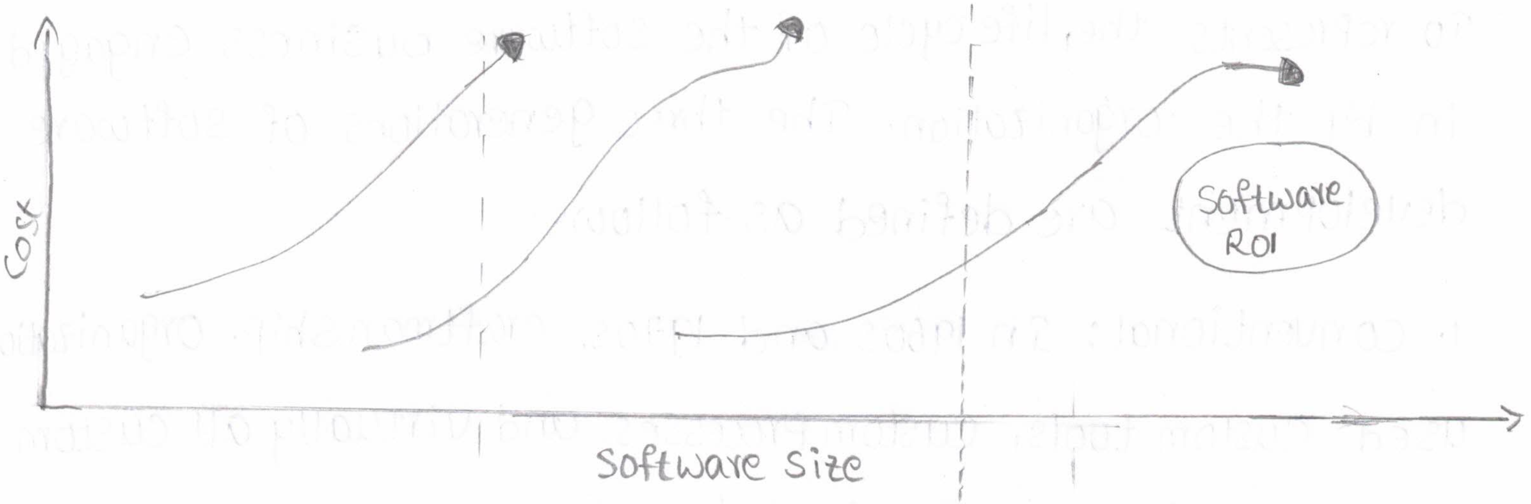
4. The environment, which is made up of the tools and techniques available to support efficient software development and to automate the Process.

5. The required quality of the Product, including its features, Performance, reliability and adaptability.

The releationships among these Parameters and the estimated cost can be written as follows:

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

Target Objective: Improved ROI



- 1960s - 1970s	- 1980s - 1990s	- 2000 and on
- waterfall model	- Process impovement	- Iterative development
- functional design	- Encapsulation - based	- component - based
- Diseconomy of scale	- Diseconomy of scale.	- Return on investment

corresponding environment, size, and Process technologies

Conventional	Transition	Modern Practices.
Environments / tools: custom	Environments/Tools: off-the-shell, seperate	Environments/Tools: off-the-shell, integrated
size: 100% custom	size: 30% component 70% custom	size: 70% component 30% custom
Process: Ad hoc	Process: Repeatable.	Process: Managed/Measured

Typical Project Performance

Predictably bad	unpredictable	Predictable
Always:	Infrequently:	usually:
over budget	on budget	on budget
over schedule	on schedule	on schedule.

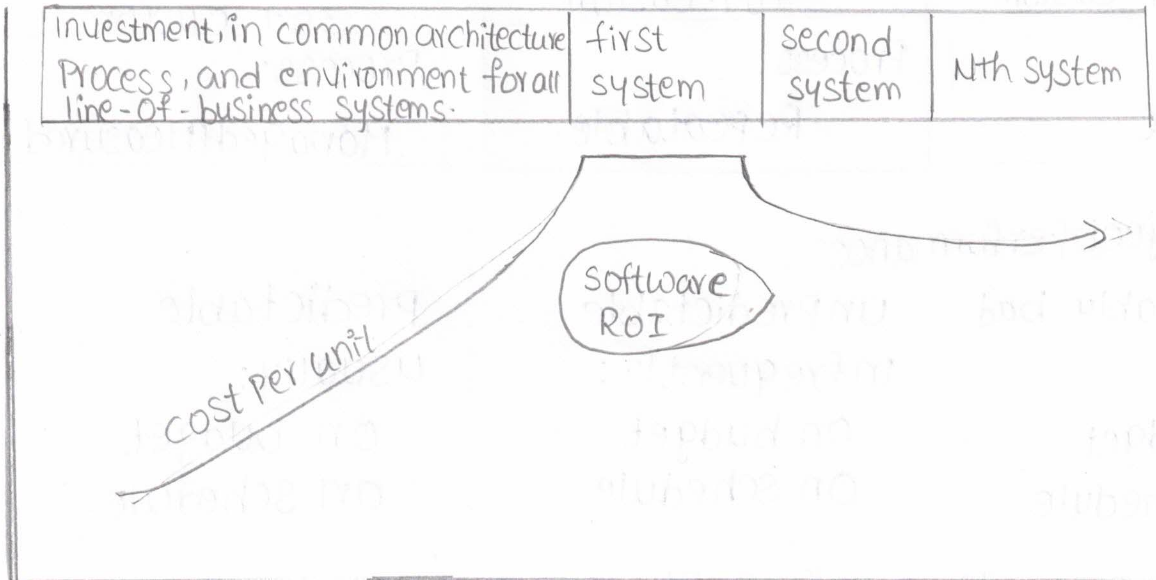
To represents the life cycle of the software business engaged in by the organization. The three generations of software deuelopment are defined as follows:

1. conuentional: In 1960s and 1970s, craftmanship. organization used custom tools, custom processes, and virtually all custom components built in primitive languages.

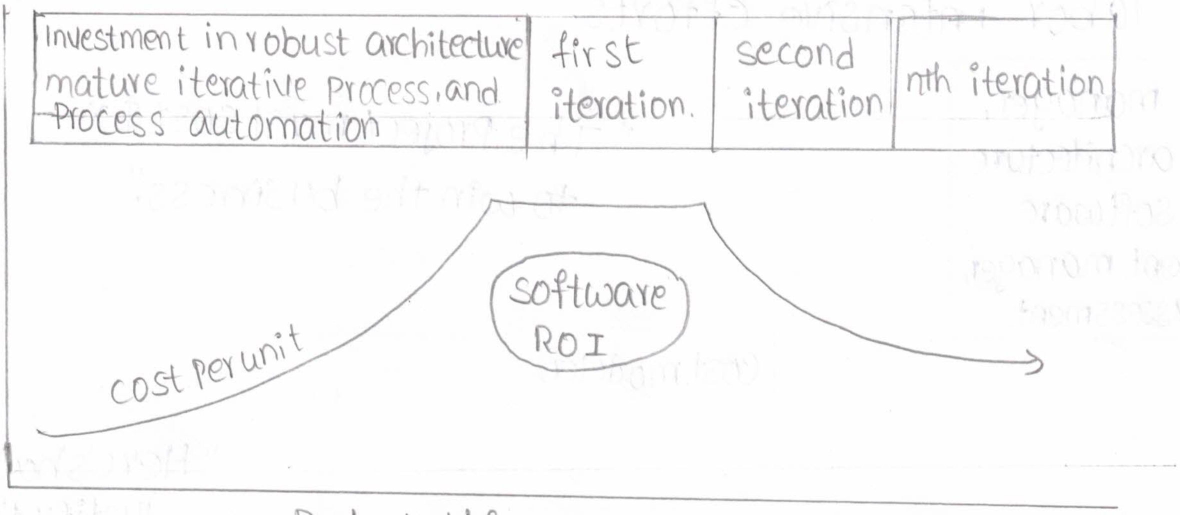
2. Transition: In 1980s and 1990s, software Engineering. organizations used more- repeatable processes and off-the shelf tools, and mostly (>70%) custom components built in higher level languages.

3. Modern Practices: In 2000 and later, software production. This book's philosophy is rooted in the use of managed and measured processes, integrated automation enivornments, and mostly (70%) off-the-shelf components.

Acheiving ROI across a line of business.

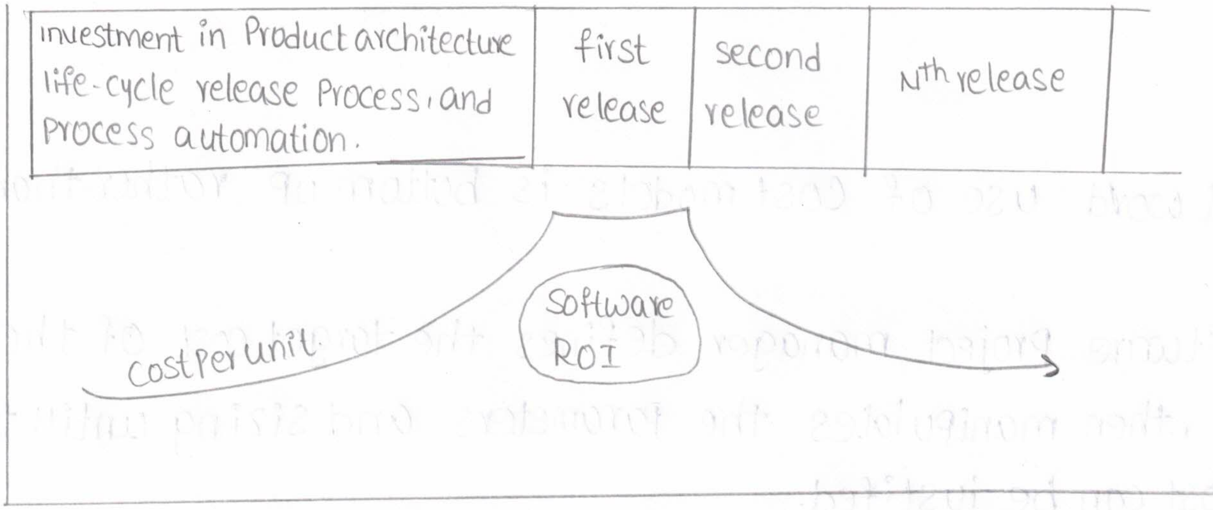

Inuestment, in common architecture process, and environment for all line-of-business systems.
first system
second system
Nth system
Software ROI
cost per unit

Acheving RoI across a Project with Multiple iterations.


Investment in robust architecture mature iterative Process, and Process automation
first iteration.
second iteration
nth iteration
Software ROI
cost per unit

Project life cycle: successive Iterations.

Acheving RoI across a life cycle of Product releases:-


investment in Product architecture life-cycle release Process, and Process automation.
first release
second release
Nth release
Software ROI
cost Per unit

Product life cycle: successive Releases.

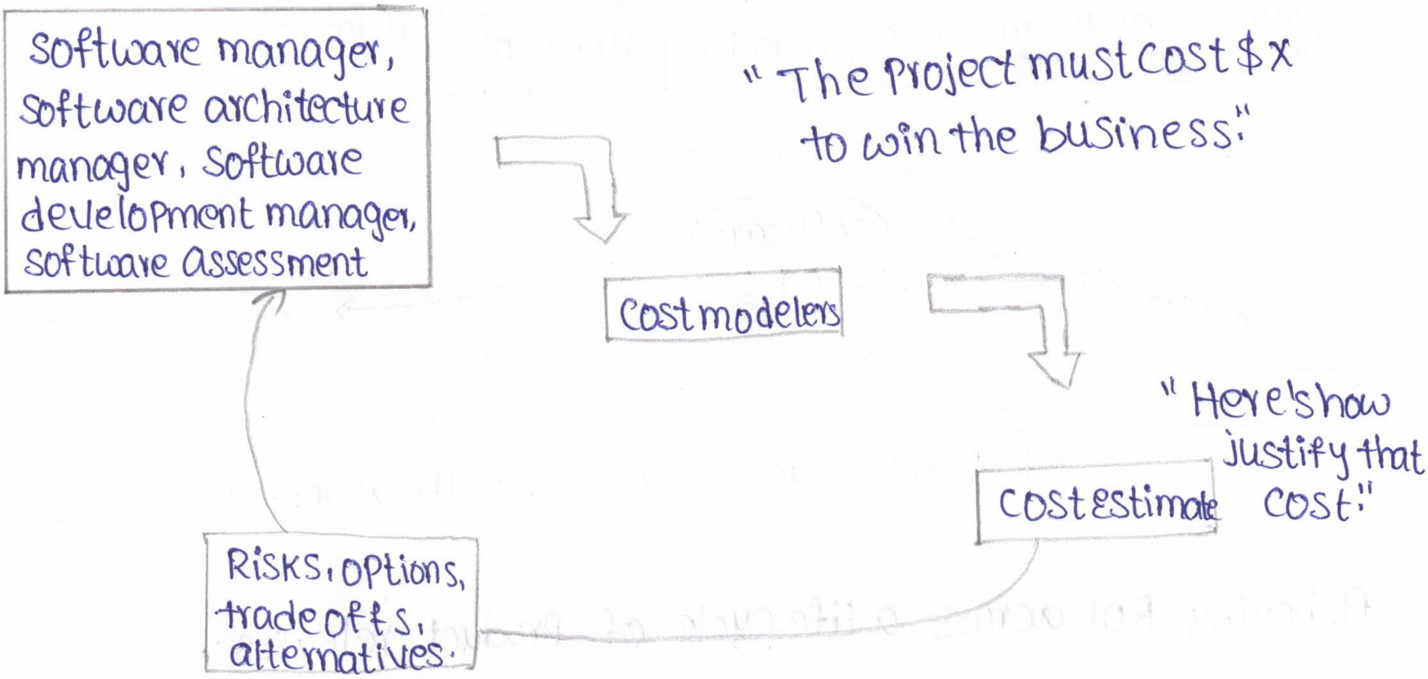
PRAGMATIC SOFTWARE COST ESTIMATION:-

→ one critical Problem in software cost Estimation is a lack of well-documentatial.

→ There have been many long-standing debates among developers and vendors of software cost Estimation models and tools.

→ The general accuracy of conventional cost model (such as COCOMO)

→ This an interesting Phenomenon to be Considered when scheduling labor-intensive efforts.


Software manager, Software architecture manager, Software development manager, Software Assessment
"The Project must cost $x to win the business."
Cost modelers
"Here's how justify that cost."
Cost estimate
Risks, options, tradeoffs, alternatives.

→ 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 untill the target cost can be justifed.

→ It is absolutely necessary to analyze the costrisks and understands the sensitivites and trade-offs Objectively.

- It is accepted by all Stakeholders as ambitious but realize.
- It is based on a well-defined Software cost model with a basis
- It is based on a database of relevant Project experience that includes Similar Process, Similar technologies, Similar quality requirments and Similar People.
- It is defined in enough detail so that its key risk areas

Improving Software Economics :-

The Economics of Software development have been not only difficult to acheive but also difficult to measure and Substaine.

The five basic Parameters of the Software cost model is.

1. Reducing the size or complexity
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.

Reducing the Software Product Size :-

→ The most significant way to improve affordability and return on investment (ROI) is usually to Produce a Product that achieves the design goals with the minimum amount of human-generated Source material.

→ To reduce the software Product size, the developers may follow Component based development model, object oriented technology, automatic code generators and so many tools.

→ The use of high level languages suchas Java, .Net, Visual Basic for Software development focused on few lines of humans generated Source code.

Languages:

To Estimate how much humans generated code is reduced to decrease the software Project Size to the Universal function Points (UFPs) are useful Estimators for language-independent.

LANGUAGE	SLOC PER UFP
Assembly	320
C	128
FORTRAN 77	105
COBOL 85	91
Ada 83	71
C++	56
Ada95	55
Java	55
Visual Basic	35.

Each Language has a domain of usage. Visual Basic is very Expressive and Powerful in building simple interactive applications.

Two interesting observations within the data concern the differences and relationships between Ada 83 and Ada 95 and between C and C++.

Ada 83 was due in part to the increase it would provide in Expressiveness.

Ada 95 represented a well-Planned language upgrade to

cost Model Parameters	Trends
Size Abstraction and component-based development technologies.	Higher order languages(C,Ct+, Ada9s, Java) Object oriented 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 Open systems Hardware Platform Performance Automation of coding, documents testing, analyses.
Quality Performance, reliability, accuracy	Hardware Platform performance Demonstration-based assessment

Ada language are numerous software Engineering technology advances, including language-enforced configuration control, seperation of interface and implementation, architectural control Primitives, encapsulation, concurrency support, and many others.

The one of the following Program sizes would be required:

1,000,000 lines of assembly language

400,000 lines of C

220,000 lines of Ada 83

175,000 lines of Ada 95 or C++.

Object - Oriented Methods and Visual Modeling :-

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

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

→ An object oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that Prevent a change in one Part of the system from rending the fabric of the entire architecture.

Booch also summarized five characteristics of a sucessful Object-Oriented Project

→ 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, Encourage communication, and 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 iterative and incremental development lifecycle.

REUSE

Reusing existing components and building reusable components have been natural software Engineering activites since the earliest improvements in Programming languages.

Software design methods have always dealt implicitly with reuse in order to minimize development costs while acheiving all the others required attributes of Performance, featureset, and quality.

Common architectures, common Processes, Precedent experience, and common enivornments are all instances of reuse.

Most truly reuasble components of value are transitional

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 profitable.

The cost of developing a reusable component is not triviac.

The steep initial curve illustrates the economic obstacle to developing reusable components. It is difficult to develop a convincing business case for development unless the objective is to support reuse across many projects.

cost and schedule investments necessary to

acheive reusable components.

Commercial components :-

The use of commercial components is certainly desirable as a means of reducing custom development, it has not Proven to be straight forward in Practice.

The advantages and disadvantages of using commerical 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 maintenancefees Dependency on vendor Run-time efficiency functionality Integration not always trivial No control overup grades and Maintanence unnecessary features that Consume extra resources often inadequate reliability and stability Multiple-vendor incompatibilities.

Custom development	Complete change freedom Smaller, often Simpler implementations Often better performance Control of development and enhancement	Expensive, unpredictable development unpredictable date undefined maintenance model Often immature and fragile Single-Platform dependency Drain on expert resources.

Improving Software Processes :-

for the Software-oriented organizations, there are many Processes and SubProcesses. I use three distinct Process.

- Meta Process : An organization's Policies, Procedures, and Practices for Pursuing a Software-intensive line of business.

- MacroProcess : A Project's Policies, Procedures, and Practices for Producing a complete software Product within a certain cost, schedule, and quality constraints.

- Micro Process : A Project Team's Policies, Procedures, and Practices for achieving an artifact of the software Process.

The Three levels of Process and their attributes.

Attributes	MetaProcess	MACRO Process	MICROProcess
Subject	lineof 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	Acquistion authorities, Customers organizational Management	Software project Managers Software Engineers	SubProjectManagers softwareEngineers
Metrics	Project Predictability Revenue, market share	on budget, on schedule major milestone Success Project Scrap and rework.	on budget, on schedule major milestone 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.

IMPROVING TEAM EFFCTIVENESS

To improve the team Effiveness the teamwork is much more important than the Sum of individuals.

The team management include the following

- A well-managed Project can succed with a nominal Engineering team.
- A mismanaged Project will almost never Succeed, even with an Expert team of engineers.
- A well architected System can be built by a nominal team of softuare builders.
- A Poorly architected System will flounder even with an expert team of builders.

The following five Staffing Principles are.

1. The Principle of top talent: use better and fever People.
2. The Principle of Job matching: fit the tasks to the skills and motivation of the People available.
3. The Principle of carrer 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 har

5. The Principle of Phase out: Keeping a misfit on the team and doesn't benefit anyone.

The following are some crucial attributes of sucessful software Project managers that deserve much more attention.

Hiring skills:- Few decisions are as important as hiring decisions. Placing the right Person in the right job seems Obvious but is surprisingly hard to achieve.

Customer. interface skill:- Avoiding adversarial releationships among stake-holders is a prerequisite for sucess.

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 ito one, and decision-making skill seems obvious despite its definition.

Team. building. skill:- Teamwork requires that a manager establish trust, motivate progress, exploit eccentric Prima donnas, transistion average people into top Performers, eliminate misfits, and diverse opinions into team direction.

Selling skill:- successful Project managers must sell at stake holders on decision and Priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell acheivements against objectives. In Practice, selling requires continuous

IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

→ An important emphasis of a modern approach is to define the development and maintenance environment as a first-class artifact of the process.

→ A robust, integrated development environment must support the automation of the development process.

→ Round-trip Engineering is a term used to describe the key capability of environments that support iterative development.

→ For-ward Engineering is the automation of one Engineering artifact from another abstract representation.

→ Round-trip engineering describes the environment support needed to change an artifact freely and have other artifacts automatically changed so that consistency is maintained among the entire set of requirements, design, implementation and deployment artifacts.

- Requirements analysis and evolution activites consume 40% of life-cycle costs.
- software design activites have an impact on more than 50% of the resources.
- coding and unit testing activites consume about 50% of software development effort and schedule.
- Test activites can consume as much as 50% of Project's

- Configuration control and change management are critical section than can consume as much as 25% of resources on large-scale Project.
- Documentation activites can consume more than 30% of Project Engineering resources.
- Project management, business administration, and Progress assessment can consume as muchas 30% of Project budgets

Achieving Required Quality:-

Many of what are accepted today as software best Practices are derived from the development Process and technologies summarized.

The key Practices that improve overall software quality include the following:

Focusing on driving requirements and critical use cases early in the life cycle, focusing on requirements completeness and traceability late in the lifecycle, and focusing throughout the lifecycle on balance between requirements evolution, design evolution and Plan evolution.

Using Metrics and indicators to measure the Progress and quality of an architecture as it envolves from a high-level Prototype into a fully compliant Product.

GENERAL QUALITY IMPROVEMENTS WITH A MODERN PROCESS.

QUALITY DRIVER	CONVENTIONAL PROCESS	MODERN ITERATIVE PROCESS
Requirements mis understanding	Discovered late	Resolved early
Development risk	Unknown until late	Understood and resolved early
commercial components	Mostly unavailable	Still a quality driver, but trade off must be resolved early in the lifecycle.
change management	late in the lifecycle, chaotic and maligant	Early in lifecycle, straight forward and benign.
Design errors	Discovered late	Resolved early
Resource adequacy	unpredictable	Predictable
schedules	Over constrained	Tunable to quality, Performance,
software Process rigor	Document-based	Managed; measured and tool-supported
Test Performance	Paper-based analysis or seperate Simula-	Executing protypes, early Performance

- Providing integrated life-cycle environments that support early and continuous configuration control, change management rigorous design Methods.

- using visual modeling and higher level languages that support architectural control, abstraction, reliable program reuse, and self documentation.

- Early and continuous insight into performance issues through demonstration-based evaluations.

The typical chronology of events in performance assessment was follows.

Project Inspection :- The Proposed design was asserted to be low risk with adequte performance margin.

Initial design review :- Optimistic assessments of adequate design margin, as early benchmarks were based mostly on paper analysis or rough simulation of the critical threads.

Mid life cycle design review :- The assessments started whittling away at the margin, as early benchmarks and inital tests began exposing the optimism inherent in earlier estimates.

Integration and test :- Serious performance problems were uncovered, necesitating fundamental changes in the architecture.

PEER INSPECTIONS: A PRAGAMATIC VIEW

-) The Peer inspections are frequently overhyped as the key aspects of a quality system.

- Transitioning Engineering information from one artifact set to another, thereby assessing the consistency, feasibility, understandability, and technology constraints inherent in the Engineering artifacts.

- Major mile stone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant use cases.

- Environment tools that ensure representation rigor, consistency completeness and change control.

- Life-cycle testing for detailed insight into critical trade-off, acceptance criteria, and requirements compliance.

- Change management metrics for objective insight into multiple perspective change trends and convergence or divergence from quality and progress goals.

In all but triyial cases, architectural issues are exposed only through more rigorous engineering activities

- Analysis, Prototyping or Experimentation
- Constructing design models
- Committing the current state of the design model to an

UNIT – II

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

THE OLD WAY AND THE NEW

THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

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

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

5.**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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT

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

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

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

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

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

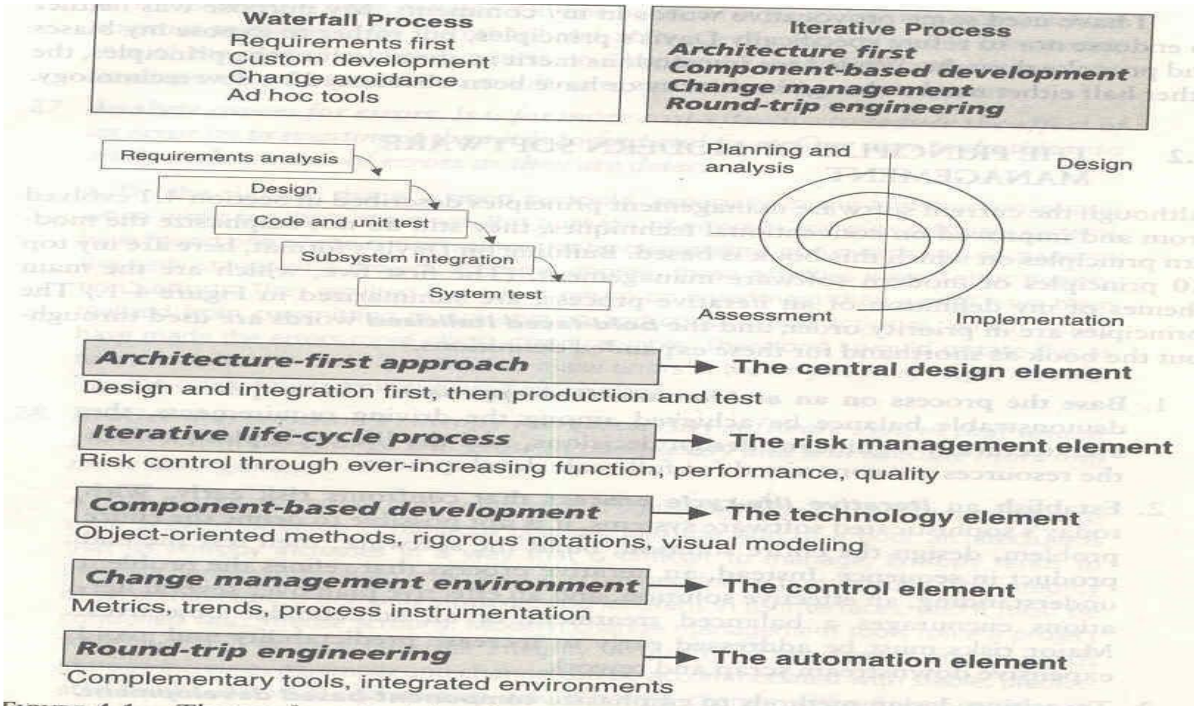

Waterfall Process
Requirements first
Custom development
Change avoidance
Ad hoc tools
Iterative Process
Architecture first
Component-based development
Change management
Round-trip engineering
Requirements analysis
Design
Code and unit test
Subsystem integration
System test
Planning and analysis
Design
Assessment
Implementation
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

FIGURE 4-1. *The top five principles of a modern process*

5. **Enhance change freedom through tools that support round-trip engineering**. Round-trip engineering is the environment support necessary to automate and synchronize engineering information in different formats(such as requirements specifications, design models, source code, executable code, test cases).
6. **Capture design artifacts in rigorous, model-based notation.** A model based approach (such as UML) supports the evolution of semantically rich graphical and textual design notations.
7. **Instrument the process for objective quality control and progress assessment**. Life-cycle assessment of the progress and the quality of all intermediate products must be integrated into the process.
8. **Use a demonstration-based approach to assess intermediate artifacts.**
9. **Plan intermediate releases in groups of usage scenarios with evolving levels of detail.** It is essential that the software management process drive toward early and continuous demonstrations within the operational context of the system, namely its use cases.
10. **Establish a configurable process that is economically scalable.** No single process is suitable for all software developments.

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

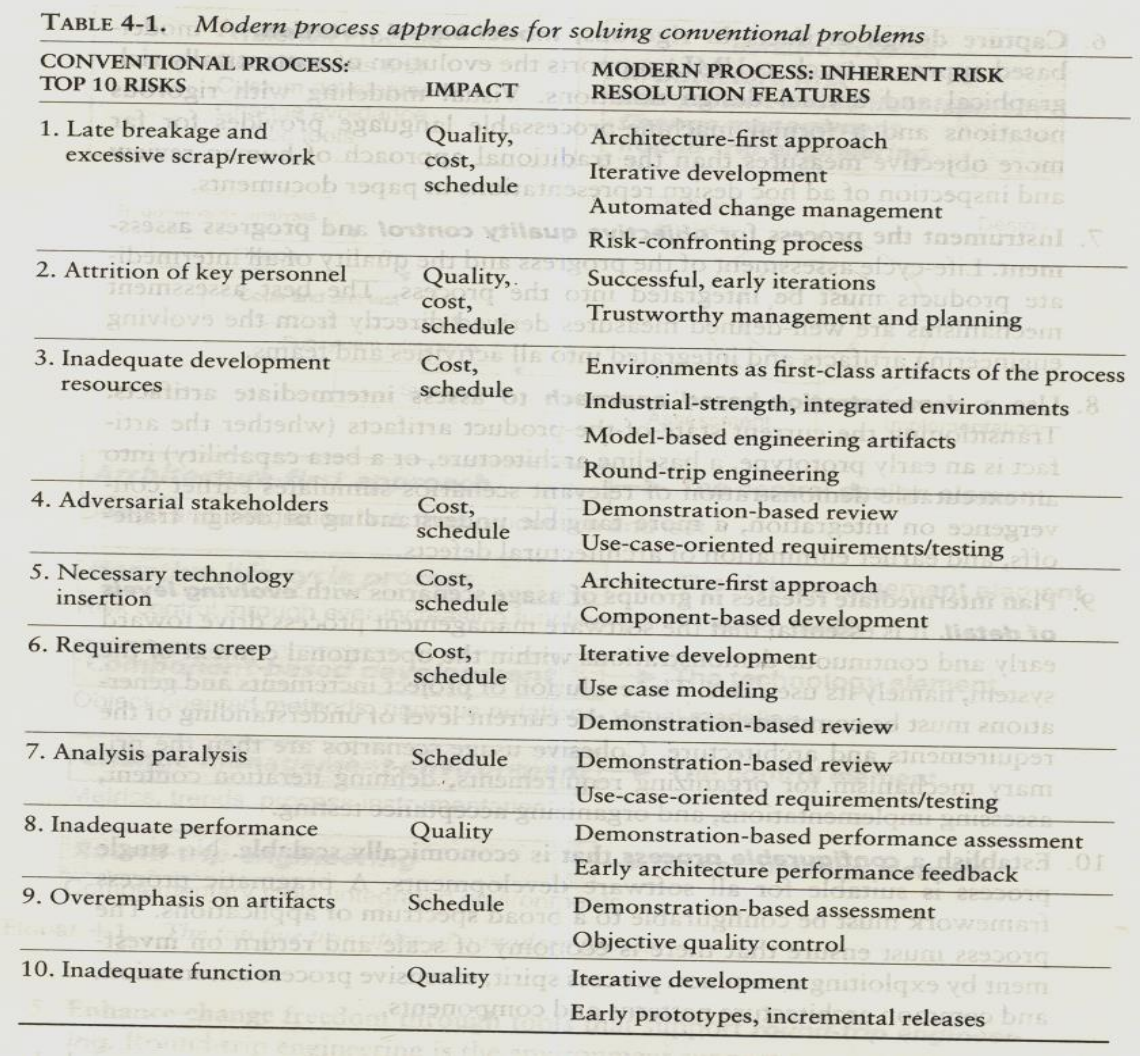


TABLE 4-1. *Modern process approaches for solving conventional problems*

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

Life cycle phases: Engineering and production stages, inception, Elaboration, construction, transition phases.

Life cycle phases

Characteristic of a successful software development process is the well-defined separation between "research and development" activities and "production" activities.

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

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

ENGINEERING AND PRODUCTION STAGES

Two stages of the life cycle are:

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

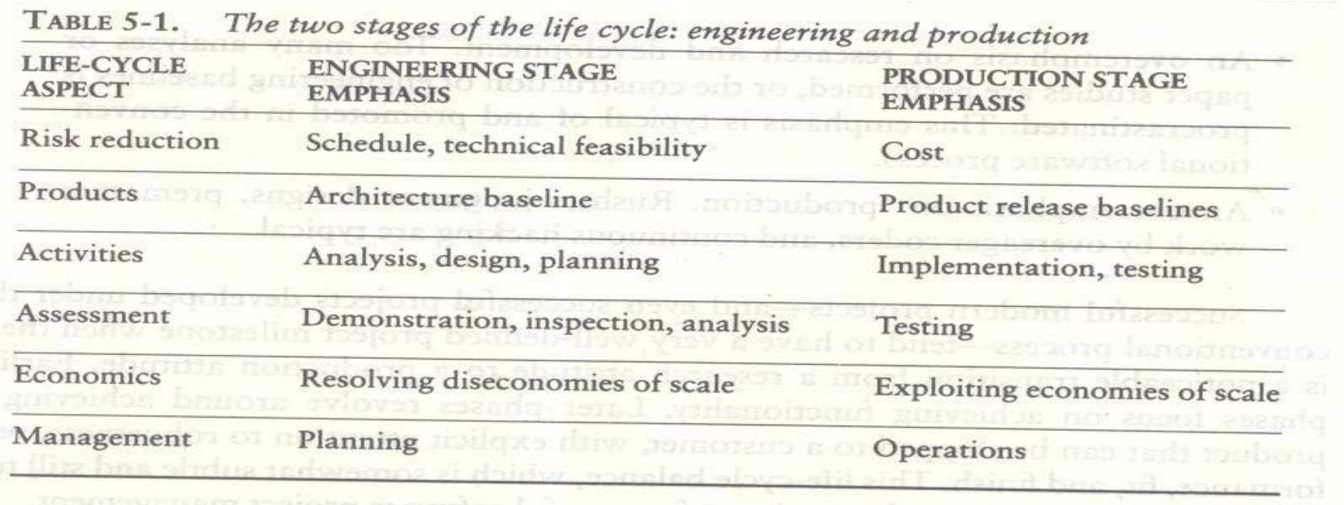


TABLE 5-1. *The two stages of the life cycle: engineering and production*

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

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

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

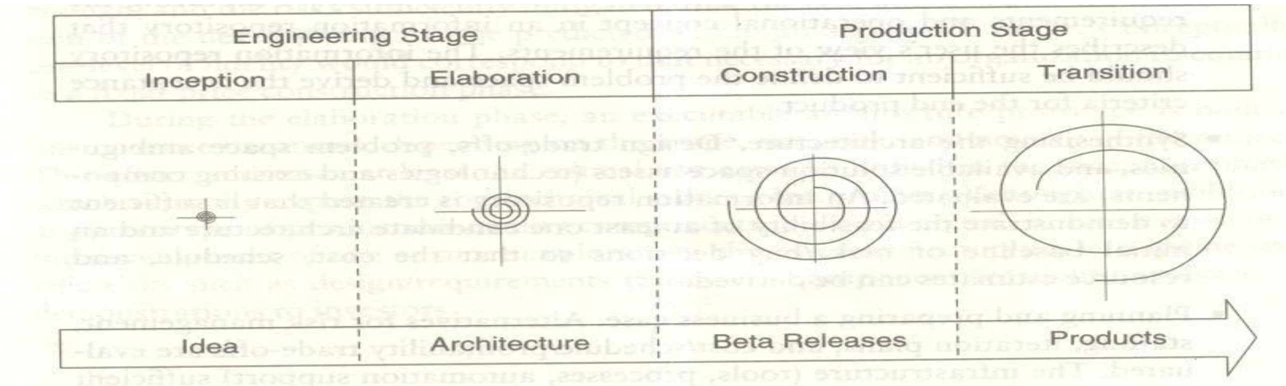


FIGURE 5-1. *The phases of the life-cycle process*

INCEPTION PHASE

PRIMARY OBJECTIVES

- Establishing the project's software scope and boundary conditions,
- Discriminating the critical use cases of the system and the primary scenarios of operations
- Demonstrating at least one candidate architecture
- Estimating the cost and schedule for the entire project (including detailed estimates for the elaboration phase)
- Estimating potential risks (sources of unpredictability)

ESSENTIAL ACTMTIES

- Formulating the scope of the project.
- The information repository should be sufficient to define the problem space and derive the acceptance criteria for the end product.
- Synthesizing the architecture.
- Planning and preparing a business case..

PRIMARY EVALUATION CRITERIA

- Do all stakeholders concur on the scope definition and cost and schedule estimates?
- Are requirements understood, as evidenced by the fidelity of the critical use cases?
- Are the cost and schedule estimates, priorities, risks, and development processes credible?
- Do the depth and breadth of an architecture prototype demonstrate the preceding criteria?

ELABORATION PHASE

PRIMARY OBJECTIVES

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

- Baselining the vision
- Baselining a high-fidelity plan for the construction phase
- Demonstrating that the baseline architecture will support the vision at a reasonable cost in a reasonable time

ESSENTIAL ACTIVITIES

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

PRIMARY EVALUATION CRITERIA

- Is the vision stable?
- Is the architecture stable?
- Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?
- Is the construction phase plan of sufficient fidelity, and is it backed up with a credible basis of estimate?
- Do all stakeholders agree that the current vision can be met if the current plan is executed to develop the complete system in the context of the current architecture?
- Are actual resource expenditures versus planned expenditures acceptable?

CONSTRUCTION PHASE

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

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES

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

PRIMARY EVALUATION CRITERIA

- Is this product baseline mature enough to be deployed in the user community? (Existing defects are not obstacles to achieving the purpose of the next release.)
- Is this product baseline stable enough to be deployed in the user community? (Pending changes are

not obstacles to achieving the purpose of the next release.)

- Are the stakeholders ready for transition to the user community?
- Are actual resource expenditures versus planned expenditures acceptable?

TRANSITION PHASE

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

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

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

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES

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

EVALUATION CRITERIA

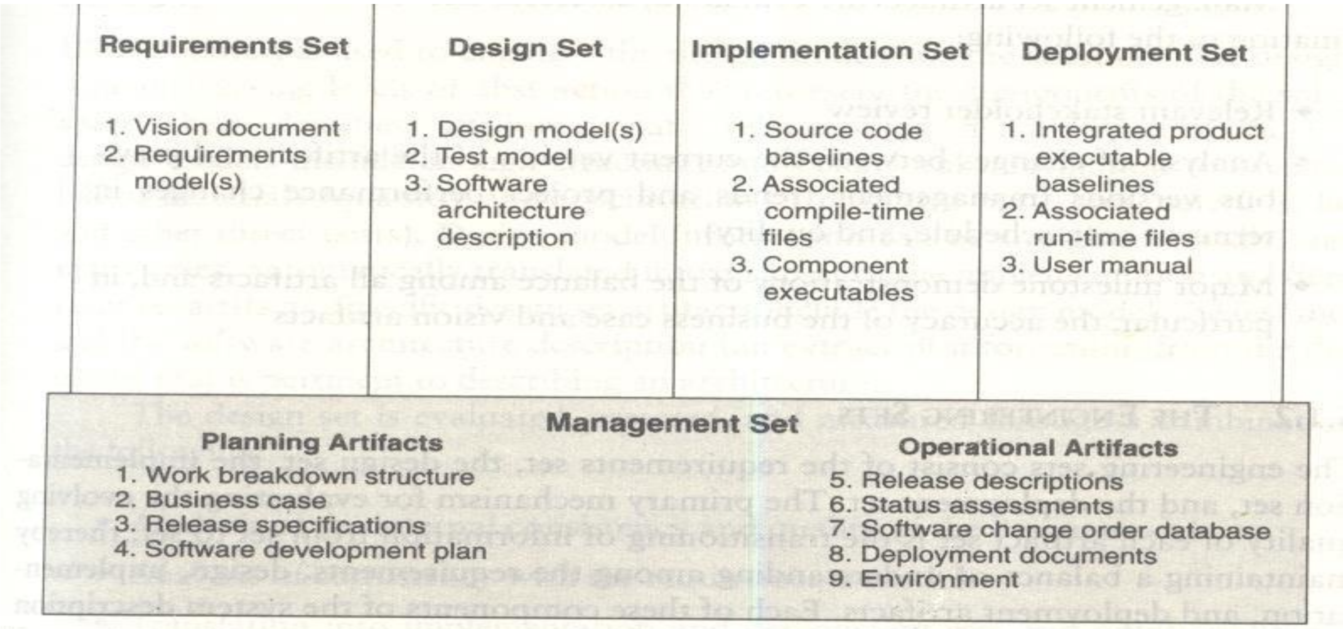
7. Is the user satisfied?
8. Are actual resource expenditures versus planned expenditures acceptable?

Artifacts of the process

THE ARTIFACT SETS

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

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



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

Management Set

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

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

THE MANAGEMENT SET

The management set captures the artifacts associated with process planning and execution.

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

- Relevant stakeholder review
- Analysis of changes between the current version of the artifact and previous versions
- Major milestone demonstrations of the balance among all artifacts and, in particular, the accuracy of the business case and vision artifacts

THE ENGINEERING SETS

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

Requirements Set

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

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

Design Set

UML notation is used to engineer the design models for the solution. The design set contains varying levels of abstraction that represent the components of the solution space (their identities, attributes, static relationships, dynamic interactions). The design set is evaluated, assessed, and measured through a combination of the following:

- Analysis of the internal consistency and quality of the design model
- Analysis of consistency with the requirements models
- Translation into implementation and deployment sets and notations (for example, traceability, source code generation, compilation, linking) to evaluate the consistency and completeness and the semantic balance between information in the sets
- Analysis of changes between the current version of the design model and previous versions (scrap, rework, and defect elimination trends)
- Subjective review of other dimensions of quality

Implementation set

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

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

- Analysis of consistency with the design models
- Translation into deployment set notations (for example, compilation and linking) to evaluate the consistency and completeness among artifact sets
- Assessment of component source or executable files against relevant evaluation criteria through inspection, analysis, demonstration, or testing
- Execution of stand-alone component test cases that automatically compare expected results with actual results
- Analysis of changes between the current version of the implementation set and previous versions (scrap, rework, and defect elimination trends)
- Subjective review of other dimensions of quality

Deployment Set

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

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

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

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

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

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

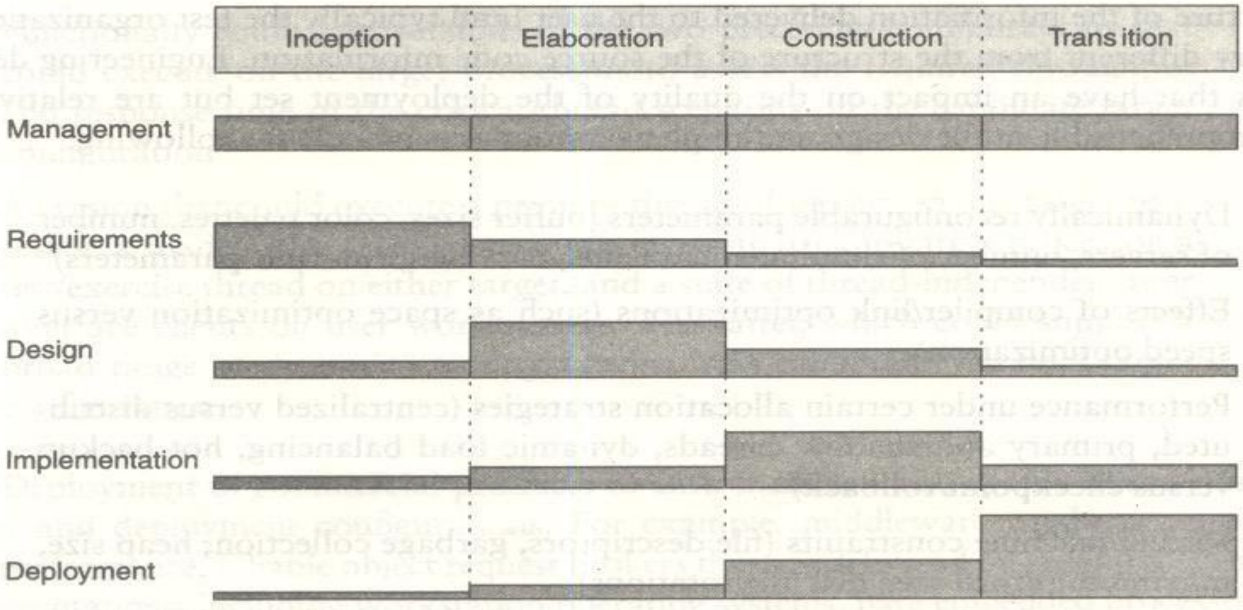


FIGURE 6-2. *Life-cycle focus on artifact sets*

Implementation Set versus Deployment Set

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

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

ARTIFACT EVOLUTION OVER THE LIFE CYCLE

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

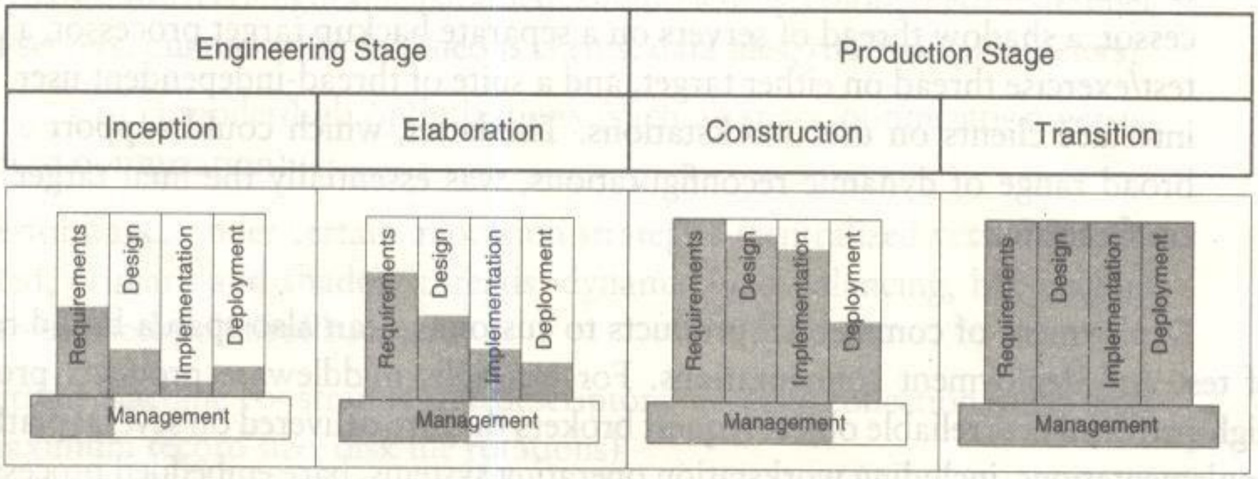


FIGURE 6-3. *Life-cycle evolution of the artifact sets*

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

TEST ARTIFACTS

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

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

- Management set. The release specifications and release descriptions capture the objectives, evaluation criteria, and results of an intermediate milestone. These artifacts are the test plans and test results negotiated among internal project teams. The software change orders capture test results (defects, testability changes, requirements ambiguities, enhancements) and the closure criteria associated with making a discrete change to a baseline.

- Requirements set. The system-level use cases capture the operational concept for the system and the acceptance test case descriptions, including the expected behavior of the system and its quality attributes. The entire requirement set is a test artifact because it is the basis of all assessment activities across the life cycle.
- Design set. A test model for nondeliverable components needed to test the product baselines is captured in the design set. These components include such design set artifacts as a seismic event simulation for creating realistic sensor data; a "virtual operator" that can support unattended, after-hours test cases; specific instrumentation suites for early demonstration of resource usage; transaction rates or response times; and use case test drivers and component stand-alone test drivers.
- Implementation set. Self-documenting source code representations for test components and test drivers provide the equivalent of test procedures and test scripts. These source files may also include human-readable data files representing certain statically defined data sets that are explicit test source files. Output files from test drivers provide the equivalent of test reports.
- Deployment set. Executable versions of test components, test drivers, and data files are provided.

MANAGEMENT ARTIFACTS

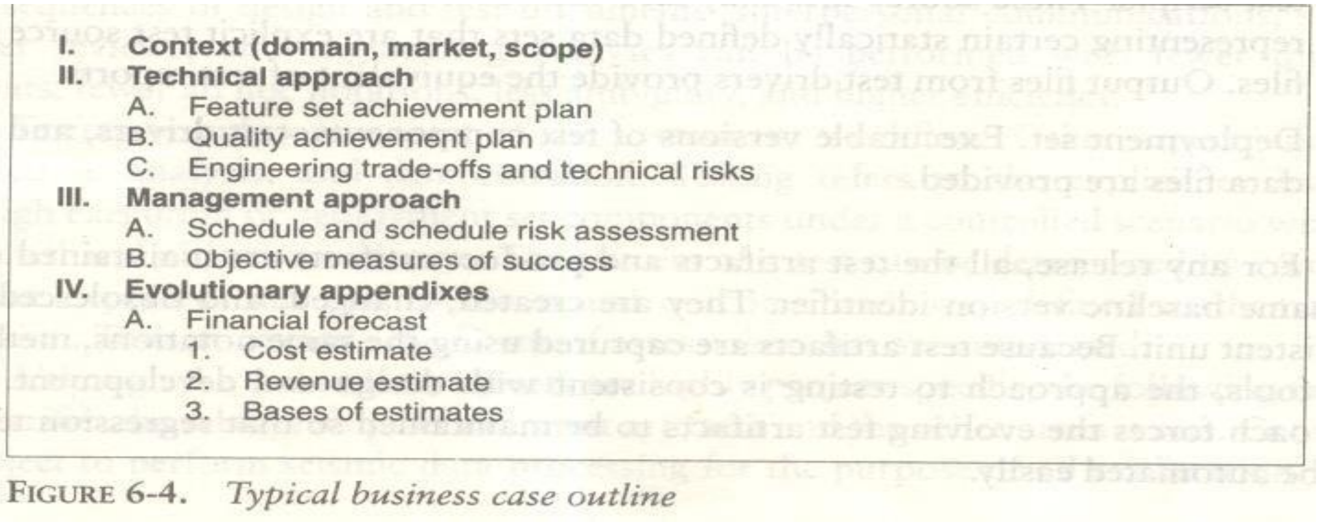
The management set includes several artifacts that capture intermediate results and ancillary information necessary to document the product/process legacy, maintain the product, improve the product, and improve the process.

Business Case

The business case artifact provides all the information necessary to determine whether the project is worth investing in. It details the expected revenue, expected cost, technical and management plans, and backup data necessary to demonstrate the risks and realism of the plans. The main purpose is to transform the vision into economic terms so that an organization can make an accurate ROI assessment. The financial forecasts are evolutionary, updated with more accurate forecasts as the life cycle progresses. Figure 6-4 provides a default outline for a business case.

Software Development Plan

The software development plan (SDP) elaborates the process framework into a fully detailed plan. Two indications of a useful SDP are periodic updating (it is not stagnant shelfware) and understanding and acceptance by managers and practitioners alike. Figure 6-5 provides a default outline for a software development plan.



I. Context (domain, market, scope)

II. Technical approach

A. Feature set achievement plan

B. Quality achievement plan

C. Engineering trade-offs and technical risks

III. Management approach

A. Schedule and schedule risk assessment

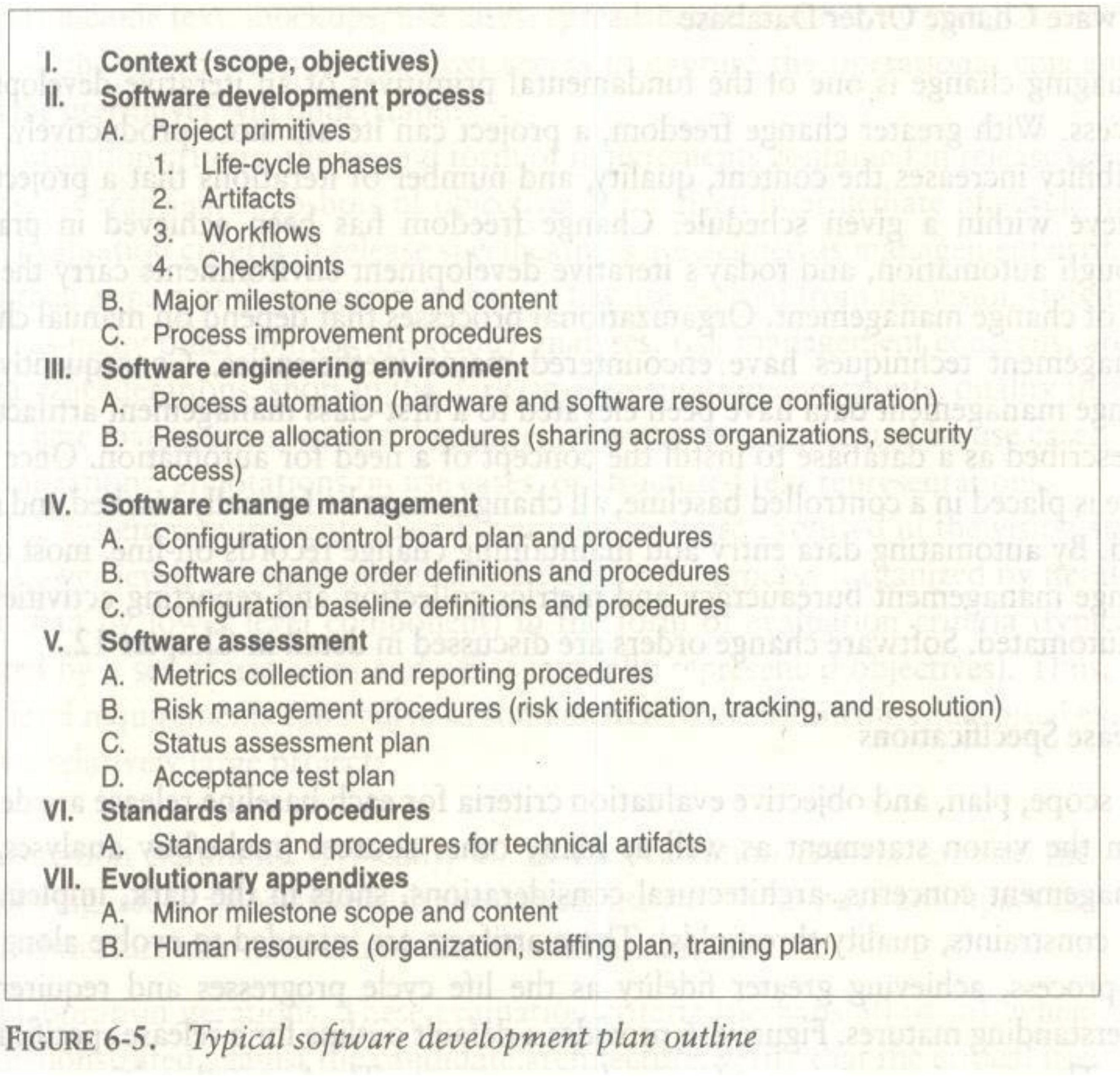
B. Objective measures of success

IV. Evolutionary appendixes

A. Financial forecast

1. Cost estimate
2. Revenue estimate
3. Bases of estimates

FIGURE 6-4. *Typical business case outline*



I. Context (scope, objectives)

II. Software development process

A. Project primitives

1. Life-cycle phases
2. Artifacts
3. Workflows
4. Checkpoints

B. Major milestone scope and content

C. Process improvement procedures

III. Software engineering environment

A. Process automation (hardware and software resource configuration)

B. Resource allocation procedures (sharing across organizations, security access)

IV. Software change management

A. Configuration control board plan and procedures

B. Software change order definitions and procedures

C. Configuration baseline definitions and procedures

V. Software assessment

A. Metrics collection and reporting procedures

B. Risk management procedures (risk identification, tracking, and resolution)

C. Status assessment plan

D. Acceptance test plan

VI. Standards and procedures

A. Standards and procedures for technical artifacts

VII. Evolutionary appendixes

A. Minor milestone scope and content

B. Human resources (organization, staffing plan, training plan)

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

Work Breakdown Structure

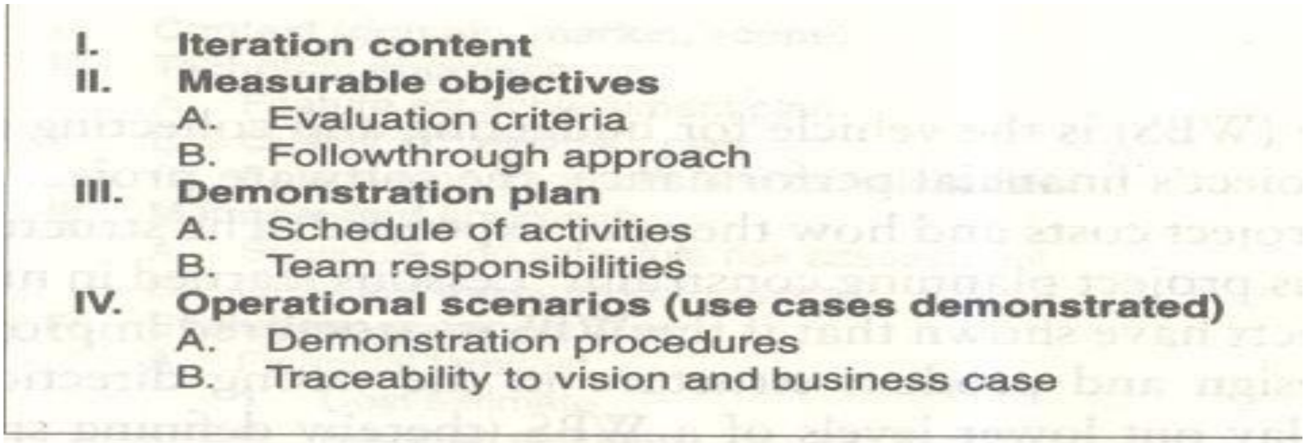
Work breakdown structure (WBS) is the vehicle for budgeting and collecting costs. To monitor and control a project's financial performance, the software project man1ger must have insight into project costs and how they are expended. The structure of cost accountability is a serious project planning constraint.

Software Change Order Database

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

Release Specifications

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



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

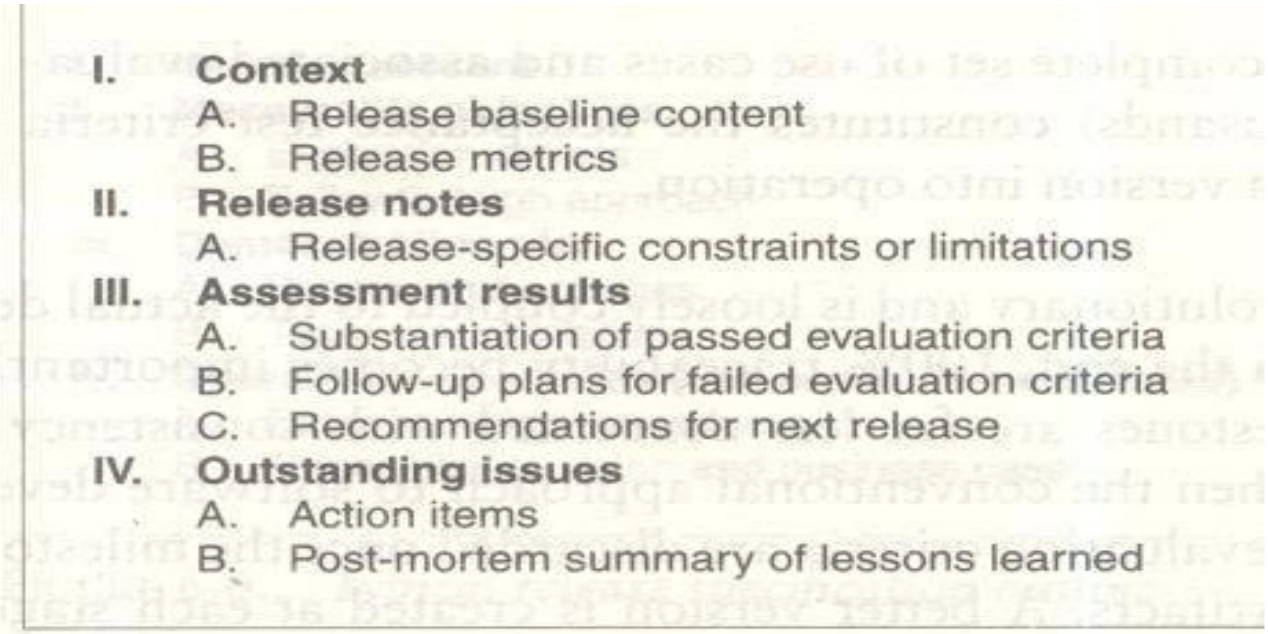
FIGURE 6-6. *Typical release specification outline*

Release Descriptions

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

Status Assessments

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



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

FIGURE 6-7. *Typical release description outline*

Environment

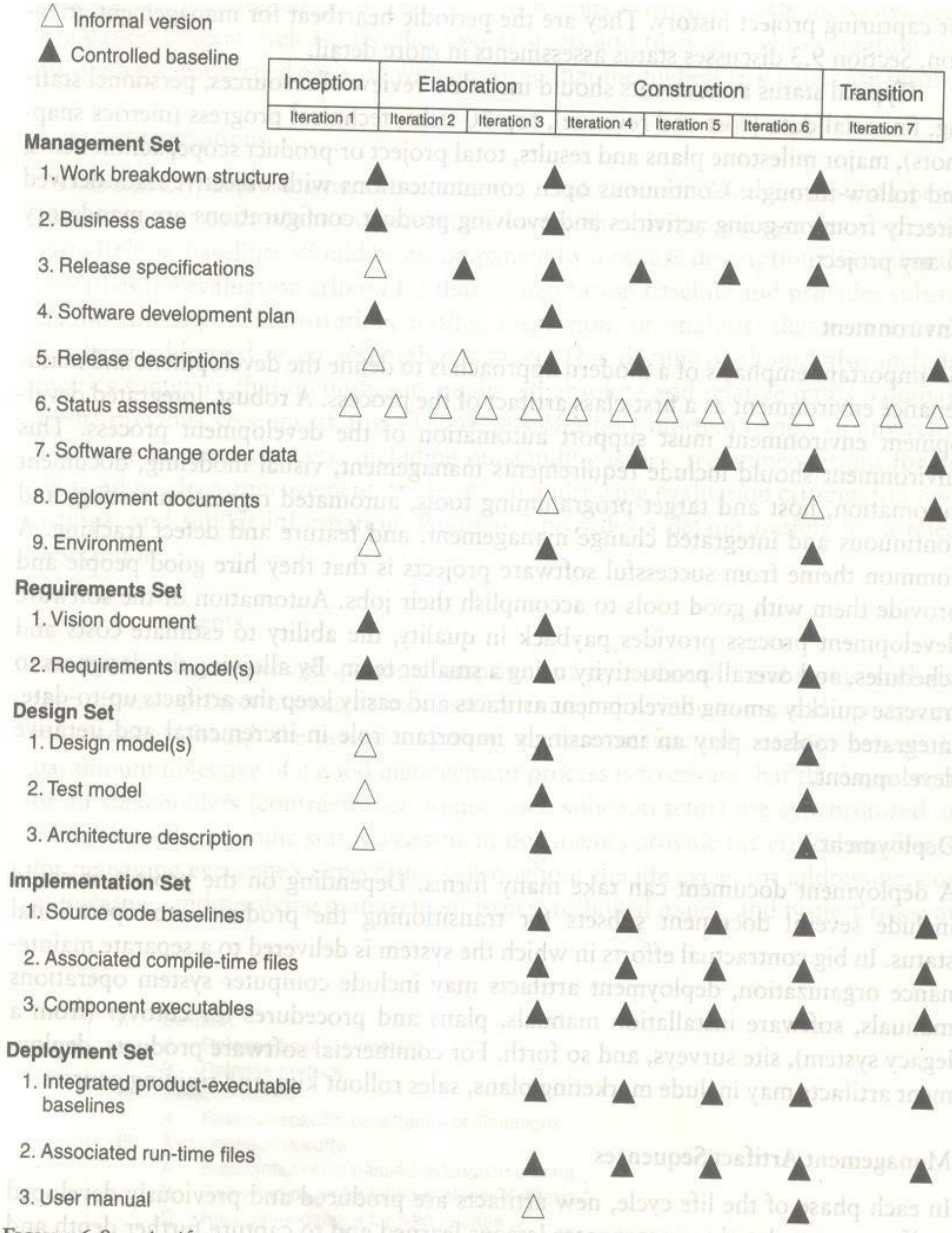
An important emphasis of a modern approach is to define the development and maintenance environment as a first-class artifact of the process. A robust, integrated development environment must support automation of the development process. This environment should include requirements management, visual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

Deployment

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

Management Artifact Sequences

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



△ Informal version

▲ Controlled baseline

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

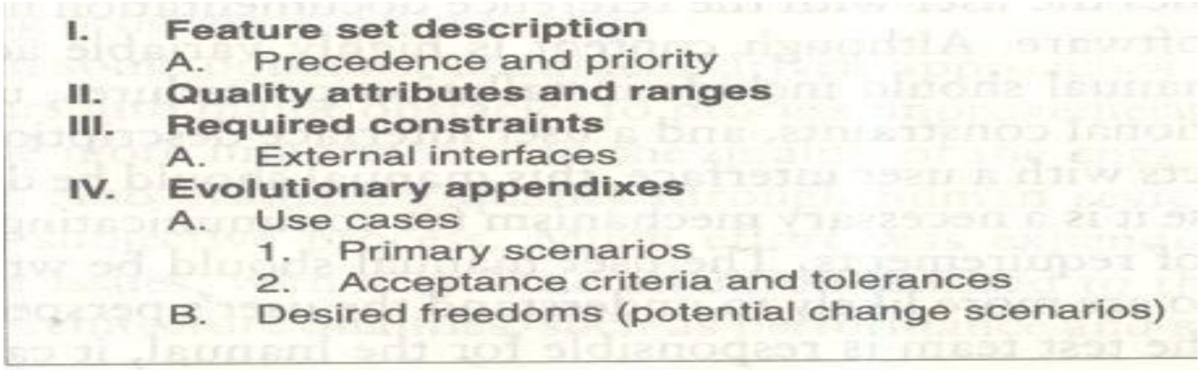
FIGURE 6-8. *Artifact sequences across a typical life cycle*

ENGINEERING ARTIFACTS

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

Vision Document

The vision document provides a complete vision for the software system under development and. supports the contract between the funding authority and the development organization. A project vision is meant to be changeable as understanding evolves of the requirements, architecture, plans, and technology. A good vision document should change slowly. Figure 6-9 provides a default outline for a vision document.

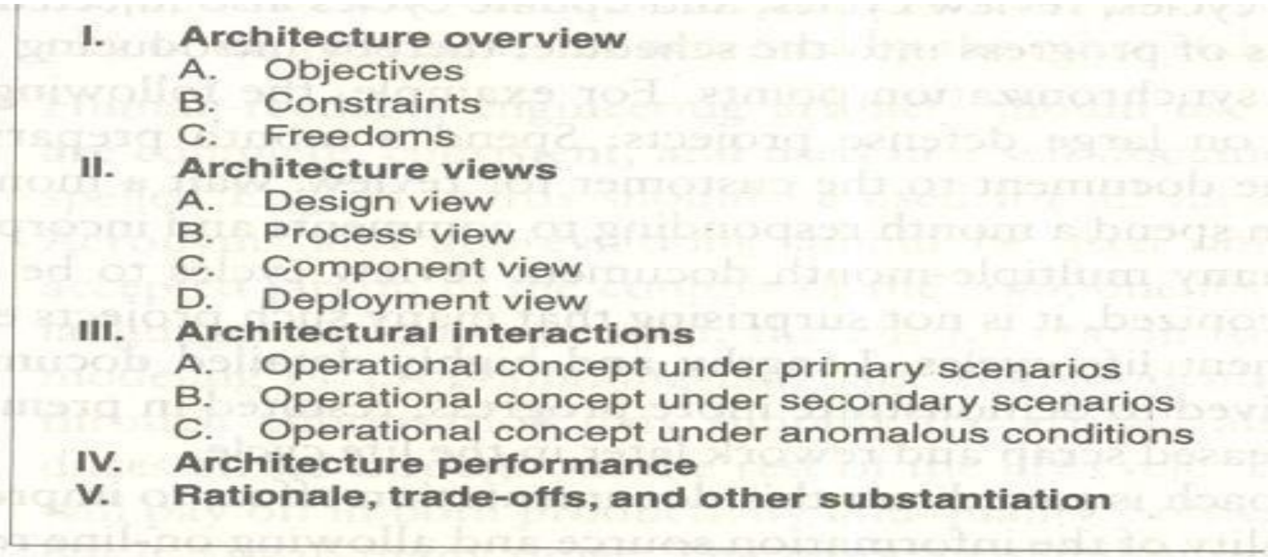


I. **Feature set description**
  A. Precedence and priority
II. **Quality attributes and ranges**
III. **Required constraints**
  A. External interfaces
IV. **Evolutionary appendixes**
  A. Use cases
    1. Primary scenarios
    2. Acceptance criteria and tolerances
  B. Desired freedoms (potential change scenarios)

FIGURE 6-9. *Typical vision document outline*

Architecture Description

The architecture description provides an organized view of the software architecture under development. It is extracted largely from the design model and includes views of the design, implementation, and deployment sets sufficient to understand how the operational concept of the requirements set will be achieved. The breadth of the architecture description will vary from project to project depending on many factors. Figure 6-10 provides a default outline for an architecture description.



I. **Architecture overview**
  A. Objectives
  B. Constraints
  C. Freedoms
II. **Architecture views**
  A. Design view
  B. Process view
  C. Component view
  D. Deployment view
III. **Architectural interactions**
  A. Operational concept under primary scenarios
  B. Operational concept under secondary scenarios
  C. Operational concept under anomalous conditions
IV. **Architecture performance**
V. **Rationale, trade-offs, and other substantiation**

FIGURE 6-10. *Typical architecture description outline*

Software User Manual

The software user manual provides the user with the reference documentation necessary to support the delivered software. Although content is highly variable across application domains, the user manual should include installation procedures, usage procedures and guidance, operational constraints, and a user interface description, at a minimum. For software products with a user interface, this manual should be developed early in the life cycle because it is a necessary mechanism for communicating and stabilizing an important subset of requirements. The user manual should be written by members of the test team, who are more likely to understand the user's perspective than the development team.

PRAGMATIC ARTIFACTS

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

UNIT - III

Model based software architectures: A Management perspective and technical perspective.
Work Flows of the process: Software process workflows, Iteration workflows.

7. Model based software architecture

ARCHITECTURE: A MANAGEMENT PERSPECTIVE

The most critical technical product of a software project is its architecture: the infrastructure, control, and data interfaces that permit software components to cooperate as a system and software designers to cooperate efficiently as a team. When the communications media include multiple languages and intergroup literacy varies, the communications problem can become extremely complex and even unsolvable. If a software development team is to be successful, the inter project communications, as captured in the software architecture, must be both accurate and precise

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

1. An *architecture* (the intangible design concept) is the design of a software system this includes all engineering necessary to specify a complete bill of materials.
2. An *architecture baseline* (the tangible artifacts) is a slice of information across the engineering artifact sets sufficient to satisfy all stakeholders that the vision (function and quality) can be achieved within the parameters of the business case (cost, profit, time, technology, and people).
3. An *architecture description* (a human-readable representation of an architecture, which is one of the components of an architecture baseline) is an organized subset of information extracted from the design set model(s). The architecture description communicates how the intangible concept is realized in the tangible artifacts.

The number of views and the level of detail in each view can vary widely.

The importance of software architecture and its close linkage with modern software development processes can be summarized as follows:

- Achieving 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 (requirements and constraints) and the solution space (the operational product).
- The architecture and process encapsulate many of the important (high-payoff or high-risk) communications among individuals, teams, organizations, and stakeholders.
- Poor architectures and immature processes are often given as reasons for project failures.
- A mature process, an understanding of the primary requirements, and a demonstrable architecture are important prerequisites for predictable planning.
- Architecture development and process definition are the intellectual steps that map the problem to a solution without violating the constraints; they require human innovation and cannot be automated.

ARCHITECTURE: A TECHNICAL PERSPECTIVE

An architecture framework is defined in terms of views that are abstractions of the UML models in the design set. The design model includes the full breadth and depth of information. An architecture view is an abstraction of the design model; it contains only the architecturally significant information. Most real-world systems require four views: design, process, component, and deployment. The purposes of these views are as follows:

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

Figure 7-1 summarizes the artifacts of the design set, including the architecture views and architecture description.

The requirements model addresses the behavior of the system as seen by its end users, analysts, and testers. This view is modeled statically using use case and class diagrams, and dynamically using sequence, collaboration, state chart, and activity diagrams.

- The *use case view* describes how the system's critical (architecturally significant) use cases are realized by elements of the design model. It is modeled statically using use case diagrams, and dynamically using any of the UML behavioral diagrams.
- The *design view* describes the architecturally significant elements of the design model. This view, an abstraction of the design model, addresses the basic structure and functionality of the solution. It is modeled statically using class and object diagrams, and dynamically using any of the UML behavioral diagrams.
- The *process view* addresses the run-time collaboration issues involved in executing the architecture on a distributed deployment model, including the logical software network topology (allocation to processes and threads of control), interprocess communication, and state management. This view is modeled statically using deployment diagrams, and dynamically using any of the UML behavioral diagrams.
- The *component view* describes the architecturally significant elements of the implementation set. This view, an abstraction of the design model, addresses the software source code realization of the system from the perspective of the project's integrators and developers, especially with regard to releases and configuration management. It is modeled statically using component diagrams, and dynamically using any of the UML behavioral diagrams.
- The *deployment view* addresses the executable realization of the system, including the allocation of logical processes in the distribution view (the logical software topology) to physical resources of the deployment network (the physical system topology). It is modeled statically using deployment diagrams, and dynamically using any of the UML behavioral diagrams.

Generally, an architecture baseline should include the following:

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

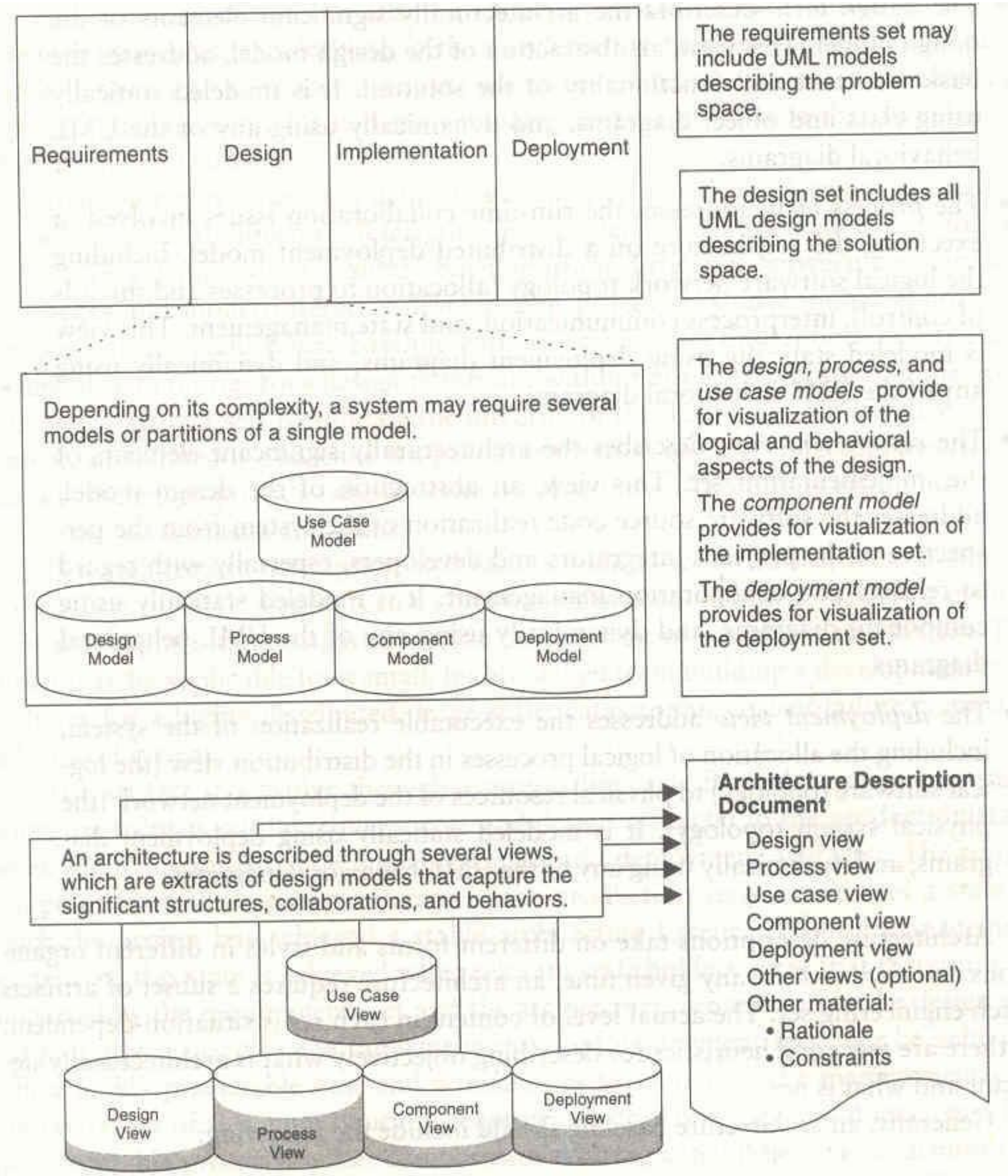

Requirements
Design
Implementation
Deployment
The requirements set may include UML models describing the problem space.
The design set includes all UML design models describing the solution space.
Depending on its complexity, a system may require several models or partitions of a single model.
Use Case Model
Design Model
Process Model
Component Model
Deployment Model
The design, process, and use case models provide for visualization of the logical and behavioral aspects of the design.
The component model provides for visualization of the implementation set.
The deployment model provides for visualization of the deployment set.
An architecture is described through several views, which are extracts of design models that capture the significant structures, collaborations, and behaviors.
Architecture Description Document
Design view
Process view
Use case view
Component view
Deployment view
Other views (optional)
Other material:
• Rationale
• Constraints
Use Case View
Design View
Process View
Component View
Deployment View

FIGURE 7-1. *Architecture, an organized and abstracted view into the design models*

8. **Workflow of the process**

SOFTWARE PROCESS WORKFLOWS

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

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

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

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

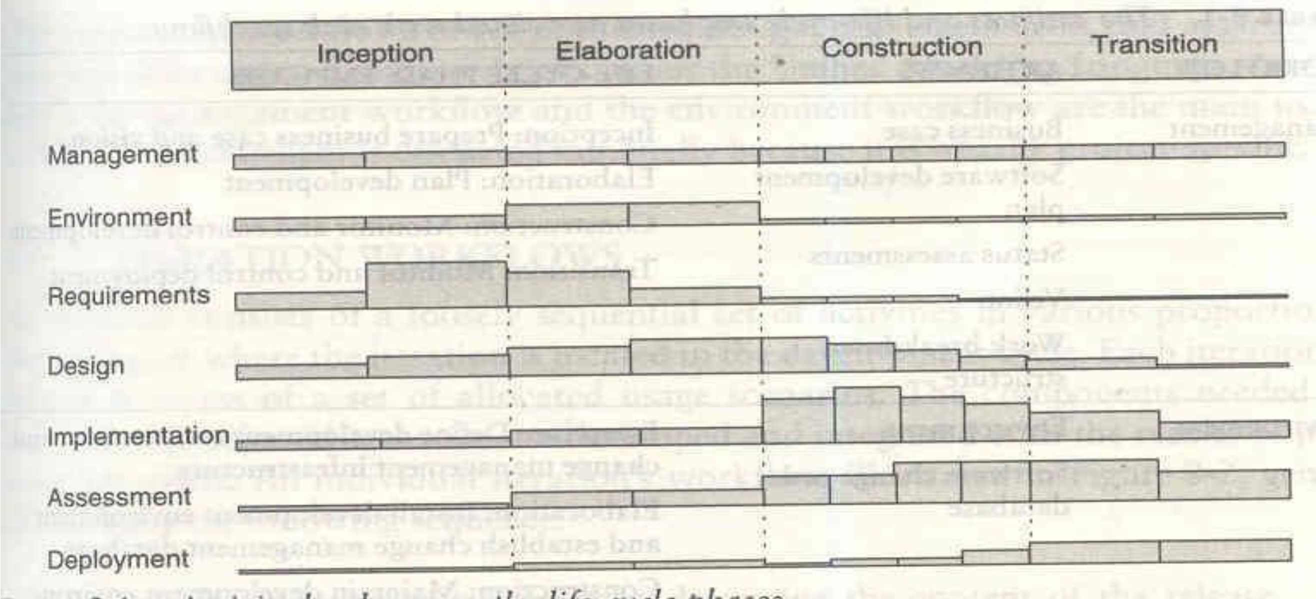


FIGURE 8-1. *Activity levels across the life-cycle phases*

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

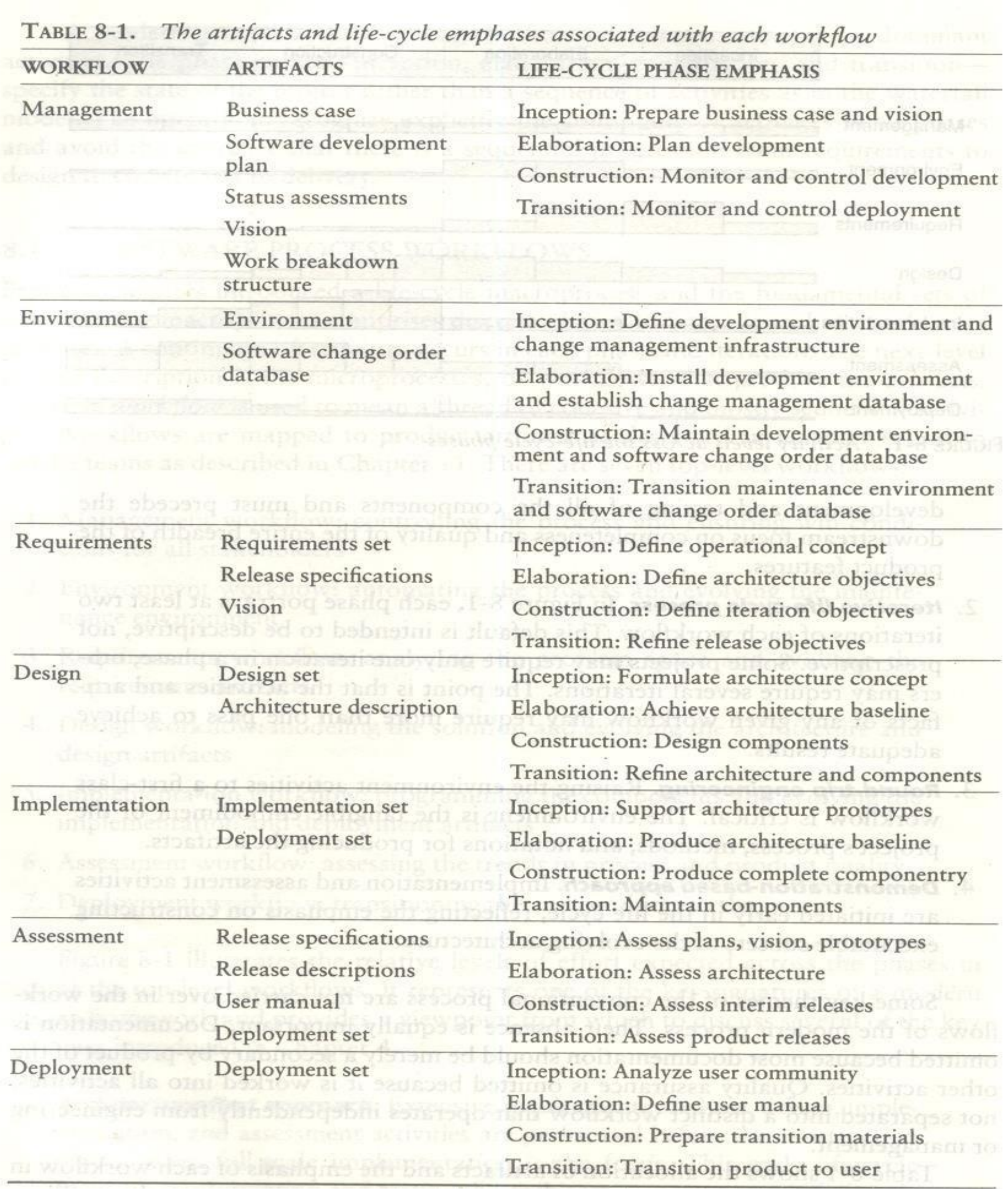


TABLE 8-1. *The artifacts and life-cycle emphases associated with each workflow*

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

ITERATION WORKFLOWS

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

- Management: iteration planning to determine the content of the release and develop the detailed plan for the iteration; assignment of work packages, or tasks, to the development team
- Environment: evolving the software change order database to reflect all new baselines and changes to existing baselines for all product, test, and environment components

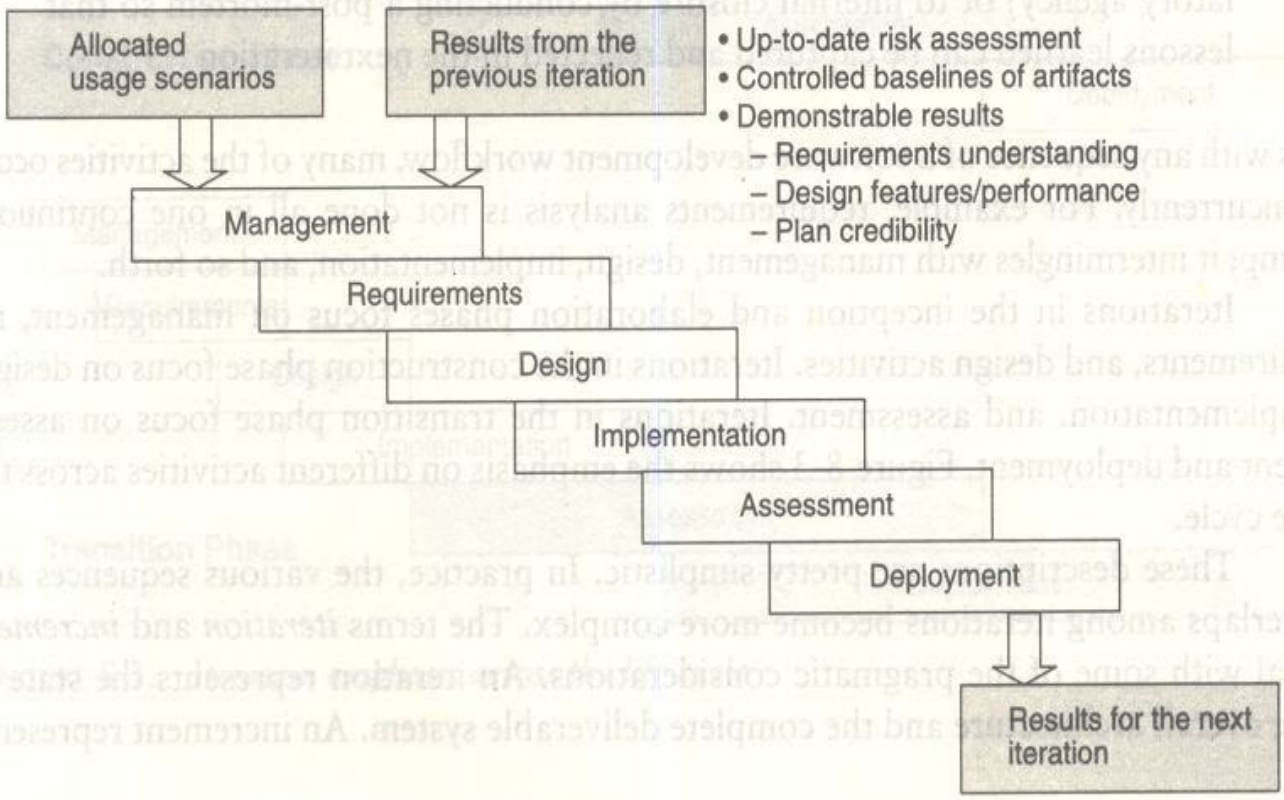

Allocated usage scenarios
Results from the previous iteration
• Up-to-date risk assessment
• Controlled baselines of artifacts
• Demonstrable results
– Requirements understanding
– Design features/performance
– Plan credibility
Management
Requirements
Design
Implementation
Assessment
Deployment
Results for the next iteration

FIGURE 8-2. *The workflow of an iteration*

- Requirements: analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to be demonstrated at the end of this iteration and their evaluation criteria; updating any requirements set artifacts to reflect changes necessitated by results of this iteration's engineering activities
- Design: evolving the baseline architecture and the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate against the evaluation criteria allocated to this iteration; updating design set artifacts to reflect changes necessitated by the results of this iteration's engineering activities
- Implementation: developing or acquiring any new components, and enhancing or modifying any existing components, to demonstrate the evaluation criteria allocated to this iteration; integrating and testing all new and modified components with existing baselines (previous versions)

- Assessment: evaluating the results of the iteration, including compliance with the allocated evaluation criteria and the quality of the current baselines; identifying any rework required and determining whether it should be performed before deployment of this release or allocated to the next release; assessing results to improve the basis of the subsequent iteration's plan
- Deployment: transitioning the release either to an external organization (such as a user, independent verification and validation contractor, or regulatory agency) or to internal closure by conducting a post-mortem so that lessons learned can be captured and reflected in the next iteration

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

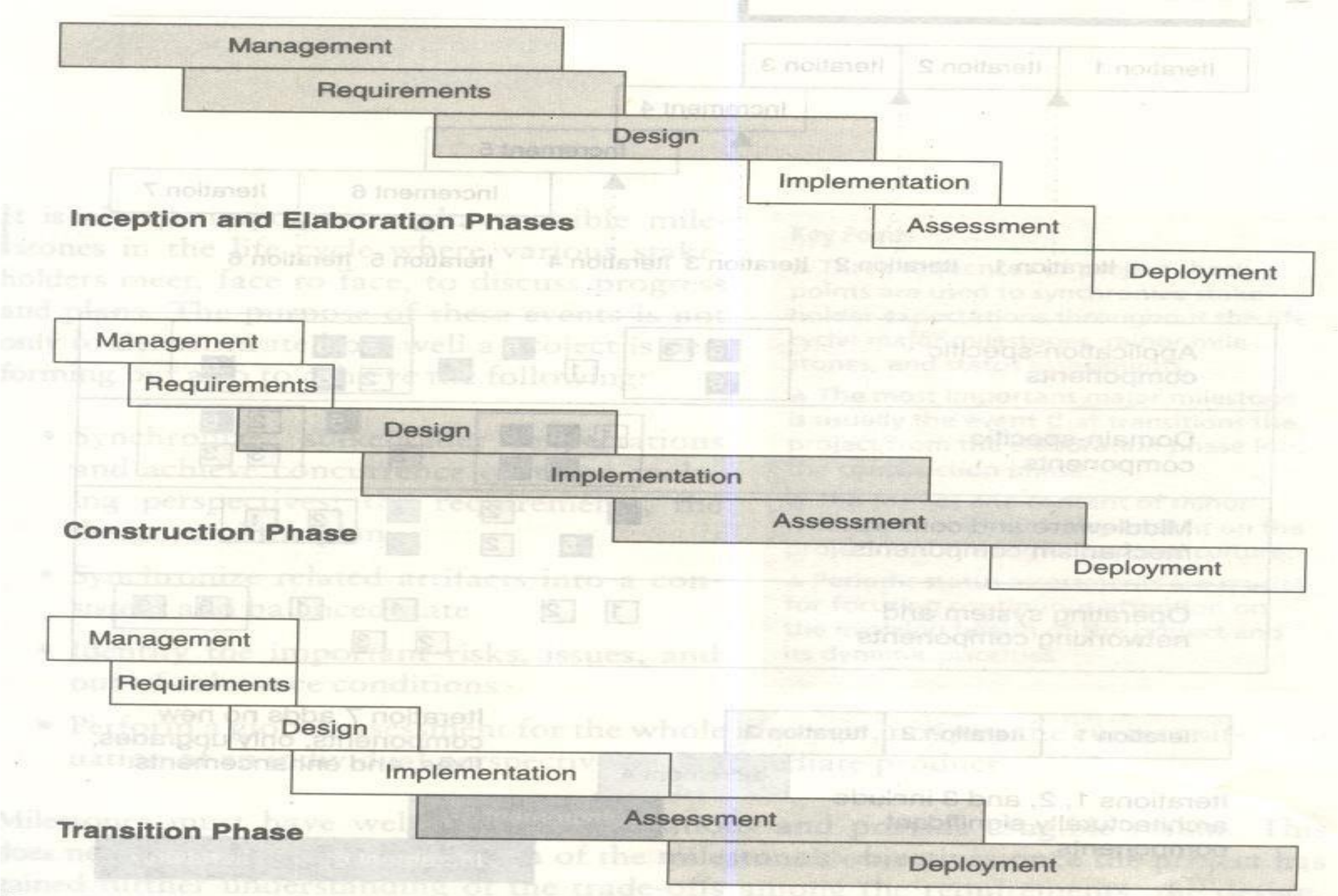
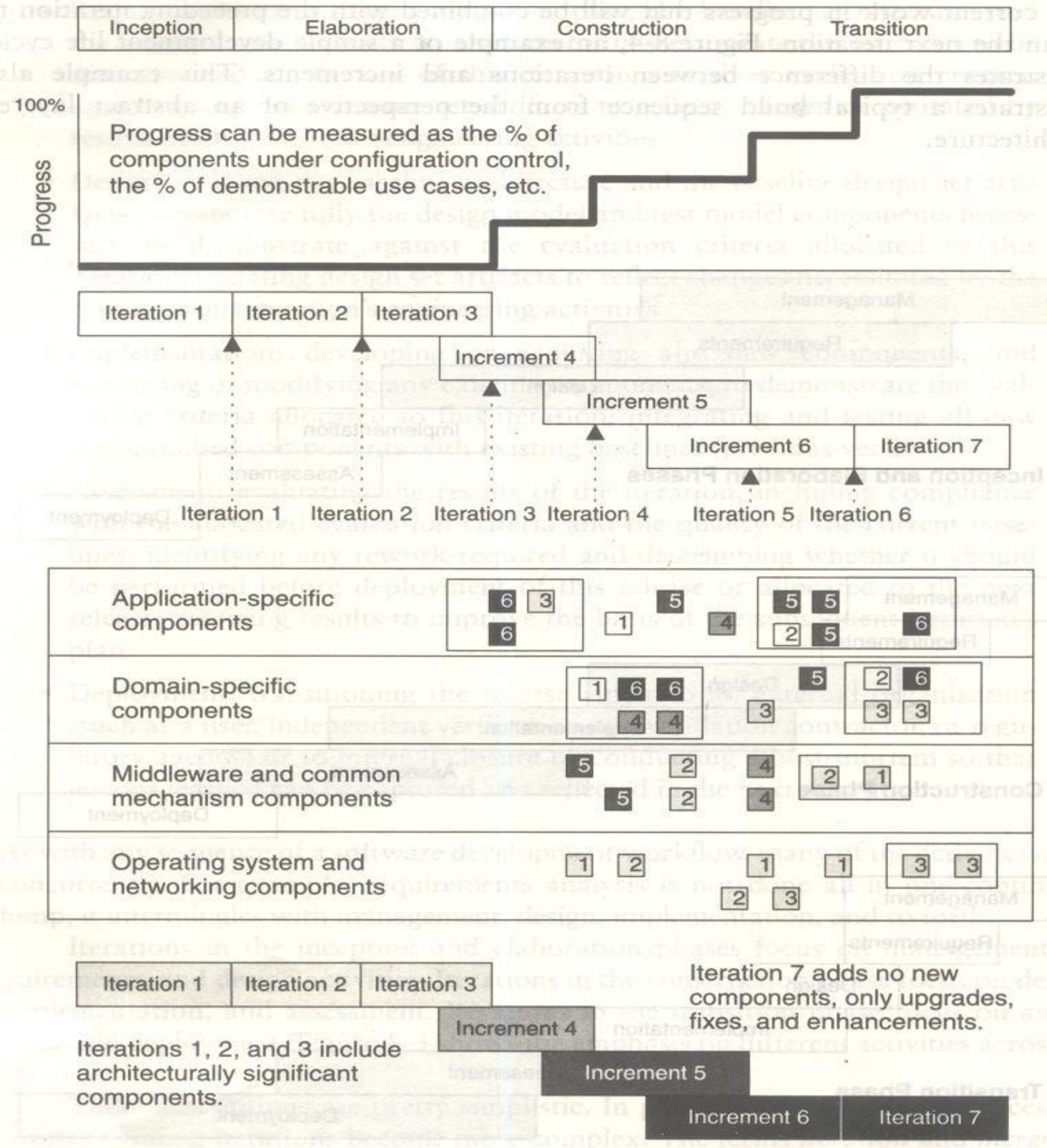


FIGURE 8-3. *Iteration emphasis across the life cycle*



Inception
Elaboration
Construction
Transition
100%
Progress
Progress can be measured as the % of components under configuration control, the % of demonstrable use cases, etc.
Iteration 1
Iteration 2
Iteration 3
Increment 4
Increment 5
Increment 6
Iteration 7
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Application-specific components
Domain-specific components
Middleware and common mechanism components
Operating system and networking components
1
2
3
4
5
6
Iteration 1
Iteration 2
Iteration 3
Increment 4
Increment 5
Increment 6
Iteration 7
Iterations 1, 2, and 3 include architecturally significant components.
Iteration 7 adds no new components, only upgrades, fixes, and enhancements.

FIGURE 8-4. *A typical build sequence associated with a layered architecture*