UNIT – 1

***Conventional Software Management** : The Waterfall Model, Conventional software Management Performance. Evolution of Software Economics: Software Economics, Pragmatic Software Cost Estimation.*

The best thing about software is its flexibility. It can be programmed to do almost anything. The worst thing about software is also its flexibility. The "almost anything" characteristic has made it difficult to plan, monitor, and control software development. This unpredictability is the basis of what has been referred to for the past 30 years as the "software crisis"

In the mid 1990's, Three important analyses of the state of the software engineering industry were performed. All three analyses reached the same general conclusion: The success rate for software projects is very low. They can be summarized as follows.

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

The above THREE analyses provide a good introduction to the magnitude of the software problem and the current norms for conventional software project management performance. Most software engineering texts present the waterfall model as the source of the conventional software process

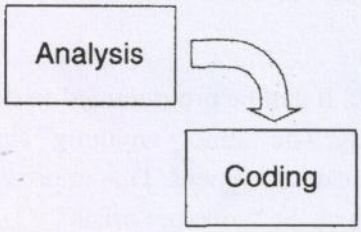
1.1 The Waterfall Model

In theory : In 1970 Winston Royce(father of walker Royce) present a paper titled "Managing the development of large scale software systems" at IEEE WESCON.

It provides 3 primary points

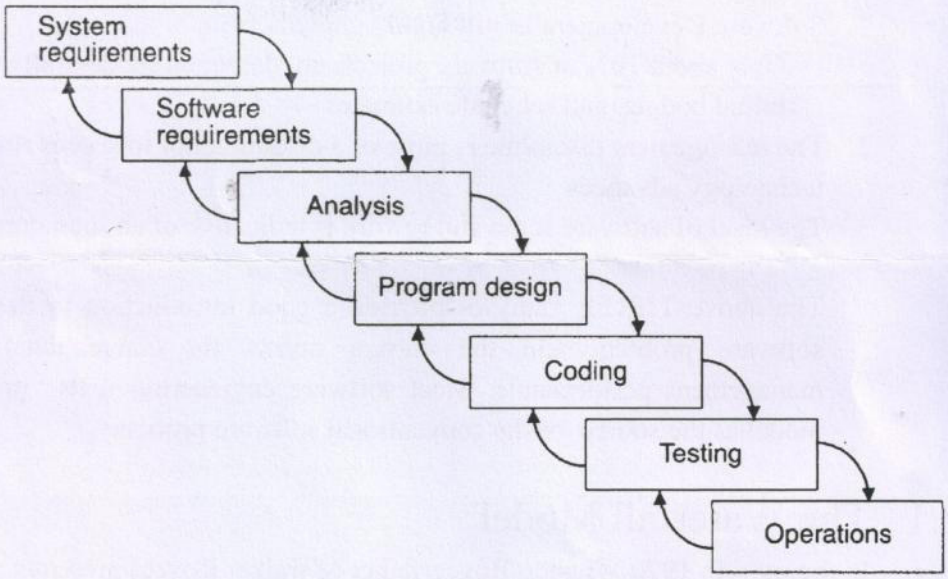
1. There are two essential steps common to the development of computer programs: **analysis and coding**
2. In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other 'overhead' steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps."
   The following fig 1.1. represents basic programming steps and large-scale approach

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



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

Waterfall Model Part 2 : The large-scale system approach



Waterfall Model Part 3 : Five necessary improvements for this approach to work

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

FIGURE 1-1. *The waterfall model*

3. **The basic framework described in the water fall model is risky and invites failure.** The testing phases that occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc. are experienced as distinguished from analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violated. Either the **requirements must be modified or a substantial design change is warranted.**

Item 1, which is seemingly trivial, will be expanded later into the separation of **engineering stage from production stage.**

To eliminate most of the development risks alluded to in item 3, the winston Royce in his paper describing five improvements to the basic waterfall model.

1. **"Program design" comes first**.

The first step toward a fix is to insert a preliminary program design phase between the software requirements generation phase and the analysis phase. By this technique , the program designer assures that the software will not fail because of storage, timing, and data flux. As analysis proceeds in the succeeding phase, the program designer must impose on the analyst the storage, timing, and operational constraints in such a way that he senses the consequences. If the total resources to be applied are insufficient or of the embryonic operational design are wrong, it will be recognized at this early stage and the iteration with requirements and preliminary design can be redone before final design, coding, and test commences. This program design procedure is implemented in the following steps.

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

Now we use the term 'architecture first' development rather than program design.

2. **Document the Design**

- Development efforts required **huge amounts** of documentation – manuals for everything
    - User manuals; operation manuals, program maintenance manuals, staff user manuals, test manuals, etc

Why do we need so much documentation?

1. Each designer MUST communicate with various stakeholders like interface designers, managers, customers, testers, developers.
2. During early phases documentation is the design
3. The real monetary value of the documentation is to support later modifications by a separate test team, maintenance team and operations personal who are not software literate

Now Visual modeling provides considerable documentation

3. Do it twice :

- History argues that the delivered version is really version #2.
- Version 1, major problems and alternatives are addressed
- Version 2, is a refinement of version 1 where the major requirements are implemented.
- Version 1 often austere; Version 2 addressed shortcomings!

Now this approach is a precursor to architecture-first development. Initial engineering is done. Forms the basis for **iterative development** and addressing **risk**!

4. Plan, Control, and Monitor Testing.

- Largest consumer of project resources (manpower, computing time, management judgment, …) is the test phase.

This is the Phase of greatest risk – in terms of cost and schedule.

This occurs last, when alternatives are least available, and expenses are at a maximum.

Typically that phase that is **shortchanged** the most

- To do:

1. Employ a team of test specialists who were not responsible for original design.

2. Employ visual inspections to spot obvious errors like code reviews, other technical reviews and interfaces, wrong address, missing factors, dropped minus signs, etc.,

3. Test every logic path

4. Employ final checkout on target computer

Now: Plan, Control, and Monitor Testing.

5. **Involve the Customer:**

Involve customer in requirements definition, preliminary software review, preliminary program design (critical design review briefings…)

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

Overall Appraisal of Waterfall Model

- Criticism of the waterfall model is misplaced.
- Theory is fine.
- Practice is what was poor!

The waterfall model in Practice :

Projects destined for trouble frequently exhibit the following symptoms

1. Protracted integration and late design breakage
2. Late risk resolution
3. Requirements-driven functional decomposition
4. Adversarial stakeholder relationships
5. Focus on documents and review meetings

1. Protracted integration and late design breakage:

The S/w was compliable and executable; it was not necessarily complete, compliant, nor up to specifications. The typical development of a S/W project that used waterfall model follows the following sequence.

- Early paper designs and thorough briefings
- Commitment to code very late in cycle
- Integration nightmares due to unforeseen implementation and interface issues
- Heavy budget and schedule pressure to get the system working

- Late 'shoe-horning' of non-optimal fixes, with no time for redesign
- A very fragile, un-maintainable product delivered late.

The following fig. shows progress profile of a conventional S/W Project

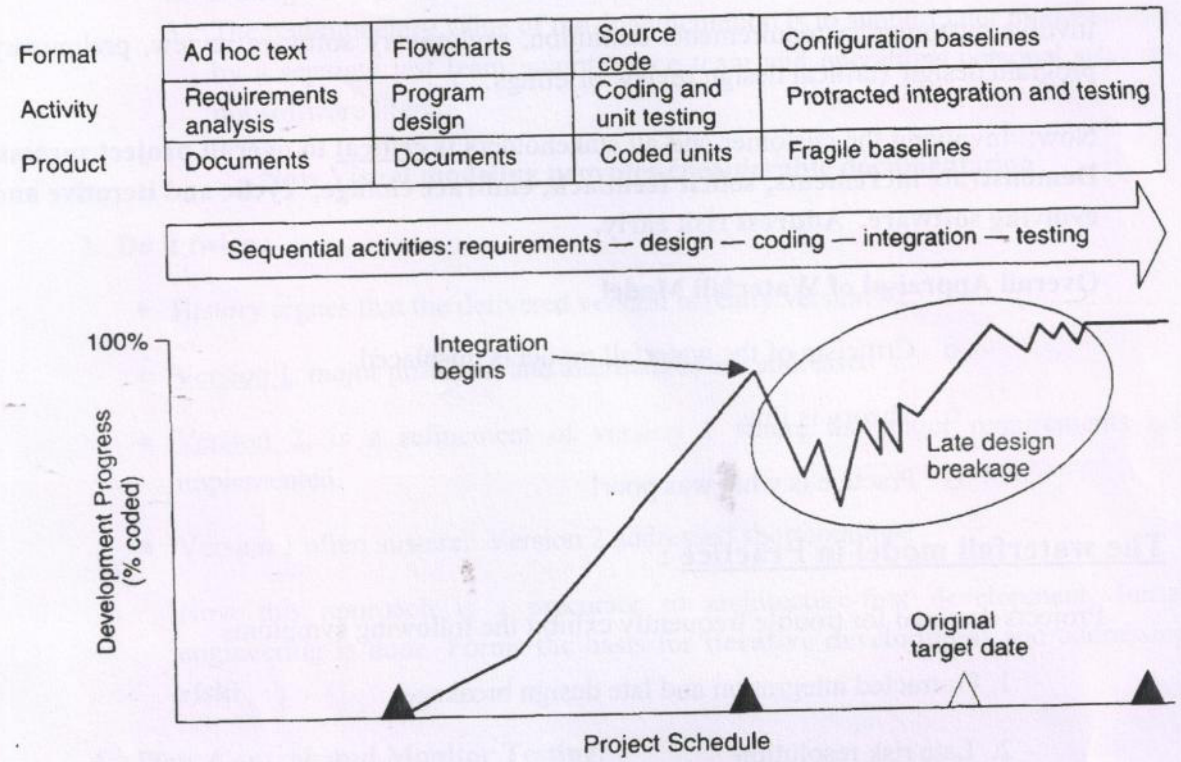


FIGURE 1-2. *Progress profile of a conventional software project*

In conventional process testing consumed 40% or more of life cycle resources. The following table shows a typical profile of cost expenditures across the spectrum of software activities.

Activity	Cost
Management	5%
Requirement	5%
Design	10%
Code and Unit Testing	30%
Integration and Testing	40%
Deployment	5%
Environment	5%
Total:	100%

2. Late risk resolution:

A serious issue associated with water fall model was the lack of early risk resolution. The following figure shows a typical risk profile for conventional water fall model projects. It includes four distinct periods of risk exposure, where risk is defined as the probability of missing a cost, schedule, feature, or quality goal.

Early in the life cycle as the requires were being specified, the actual risk exposure was highly unpredictable

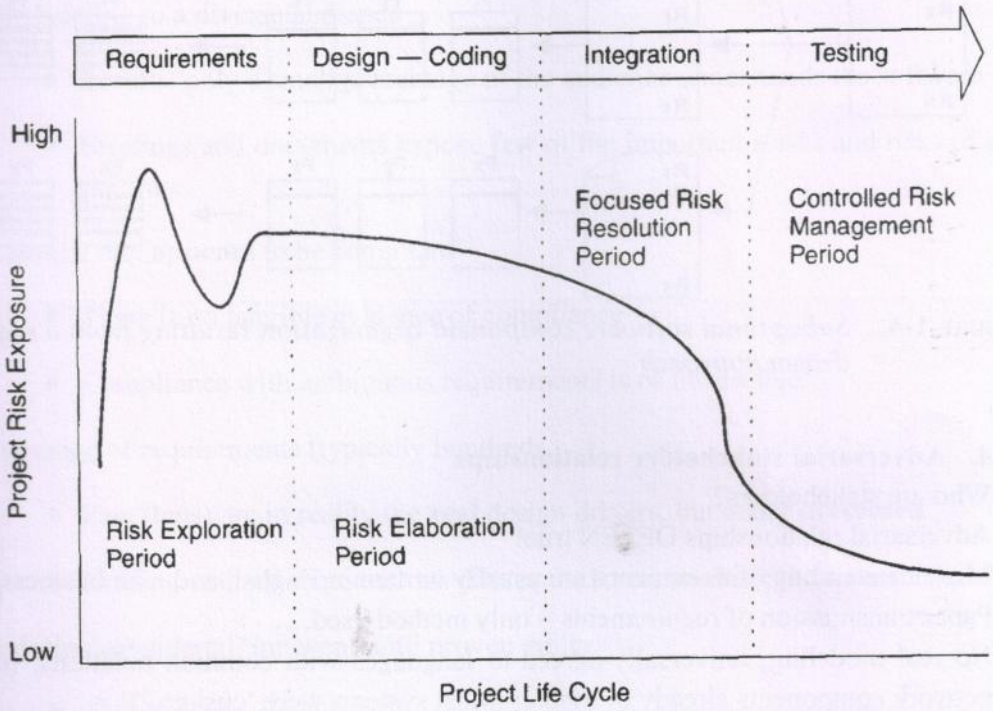


FIGURE 1-3. *Risk profile of a conventional software project across its life cycle*

3. Requirements-driven functional decomposition

- Traditionally, software development processes have been requirements-driven.
- Developers: assumed requirement specs: complete, clear, necessary, feasible, and remaining constant! This is RARELY the case!!!!
- All too often, too much time spent on equally treating 'all' requirements rather than on critical ones.
- Much time spent on documentation on topics (traceability, testability, etc.) that was later made obsolete as 'driving requirements and subsequent design understanding evolve.'

Another property of conventional approach is the requirements were specified in functional manner. The following figure shows requirements driven functional decomposition

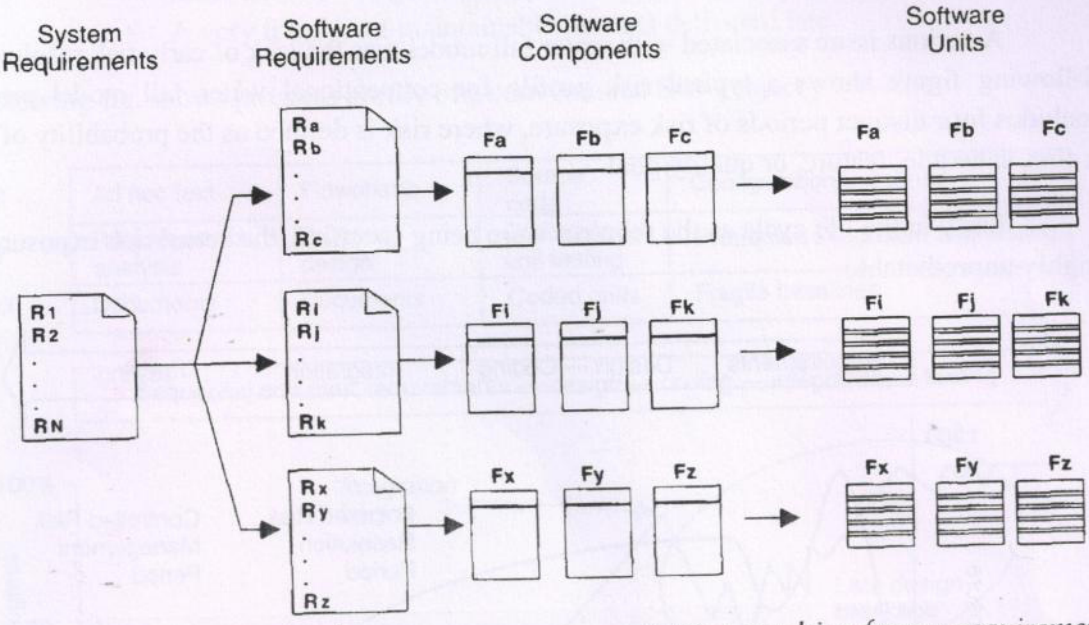


FIGURE 1-4. *Suboptimal software component organization resulting from a requirements-driven approach*

4. Adversarial stakeholder relationships

- Who are stakeholders?
- Adversarial relationships OFTEN true!
- Misunderstanding of documentation usually written in English and with business jargon.
- Paper transmission of requirements – only method used….
- No real modeling, universally-agreed-to languages with common notations; (no GUIs, network components already available; Most systems were 'custom.')
- Subjective reviews / opinions
- Management Reviews; Technical Reviews!

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

1. The Contractor prepared a draft contract-deliverable document that constituted an intermediate artifact and delivered it to the customer for approval. (usually done after interviews, questionnaires, meetings…)

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

3. The Contractor incorporated these comments and submitted (typically 15-30 days) a final version for approval.

5. Focus on documents and review meetings

The conventional process focused on producing various documents that attempted to describe the software product, with insufficient focus on producing tangible increments of the product.

Results of conventional Software product design Reviews :

1. Big briefing to a diverse audience
   - Results: only a small percentage of the audience understands the software
   - Briefings and documents expose **few** of the important assets and risks of complex software.
2. A design that **appears** to be compliant
   - There is no tangible evidence of compliance
   - Compliance with ambiguous requirements is of little value.
3. Coverage of requirements (typically hundreds….)
   - Few (tens) are in reality the **real** design drivers, but many **presented**
   - Dealing with **all** requirements dilutes the focus on **critical drivers**.
4. A design considered 'innocent until proven guilty'
   - The design is always guilty
   - Design flaws are exposed later in the life cycle.

1.2 Conventional Software Management Performance

Barry Boehm's "industrial software metrics top ten list " for S/W project management performance

1. Finding and fixing a software problem after delivery costs 100 times more than fining and fixing the problem in early design phases.
2. You can compress software development schedules 25% of nominal, but no more.
3. For every $1 you spend on development, you will spend $2 on maintenance.

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

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

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

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

8. Software systems and products typically cost three times as much per SLOC as individual software programs. Software-system products, that is system of systems, cost nine times as much.

9. Walkthroughs catch 60% of the errors

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

2.1 Evolution of Software Economics :

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

1. Size 2. Process 3. Personnel 4. Environment 5. Required Quality

1. **Size :** The size of the end product is typically quantified in-terms of no of source lines of code (SLOC) or No of Function Points (FPs)
2. **Process :** The process used to produce the end product, the ability of the process to avoid non-value adding activities like rework, bureaucratic delays, communication over head
3. **Personnel :** the capabilities of Software Engineering personnel, and particularly their experience with the computer science issues and the applications domain issues of the project.
4. **Environment :** which is made up of tools and techniques available to support efficient software development and to automate the process.
5. **Required Quality :** The required quality of the product including its features, performance, reliability, and adaptability.

The relationship among these FIVE parameters and the estimated cost can be written as

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

Three generations of software development

The following figure shows THREE generations of basic technology advancements in tools, components and processes

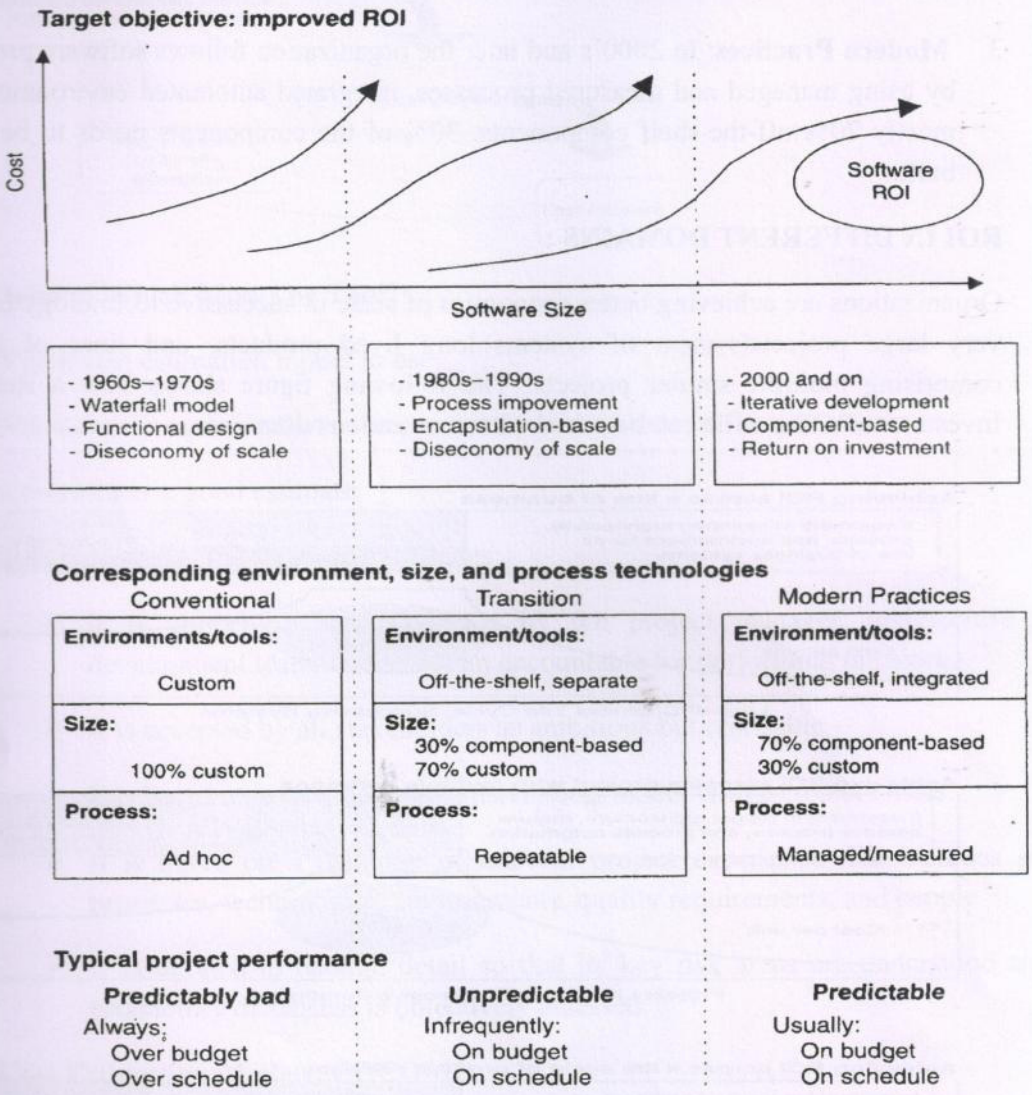


FIGURE 2-1. *Three generations of software economics leading to the target objective*

Three generations of the Software Development are defined as follows

1. **Conventional** : In 1960's – 1970's the organizations follows craftsmanship's by using custom tools, custom processes and virtually all custom components built-in primitive languages. Project performance was highly unpredictable in that cost, schedule, and quality objectives were almost always under achieved.

2. **Transition:** In 1980's – 1990's the organizations follows software Engineering by using moré repeatable process and off-the-shelf tools, and mostly (>70% custom components) built-in higher languages. Some of the components (30 %) are available as commercial components, including the OS, DBMS, Networking and graphical user interfaces.

3. **Modern Practices**: In 2000's and later the organization follows software production by using managed and measured processes, integrated automated environments, and mostly 70% off-the-shelf components. 30% of the components needs to be custom built.

ROI IN DIFFERENT DOMAINS :

Organizations are achieving better economies of scale in successive technology eras-with very large projects(system of systems),long lived products, and lines of business comprising multiple similar projects. The following figure shows how a Return On Investment(ROI) profile can be achieved across various domains.

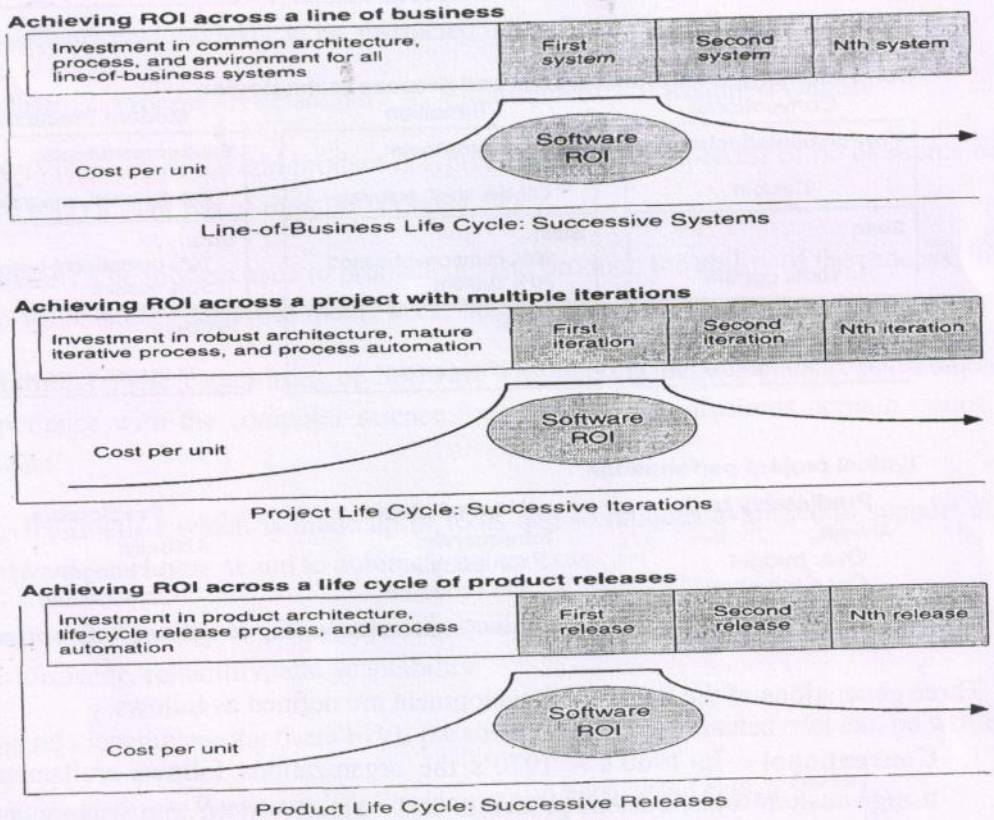
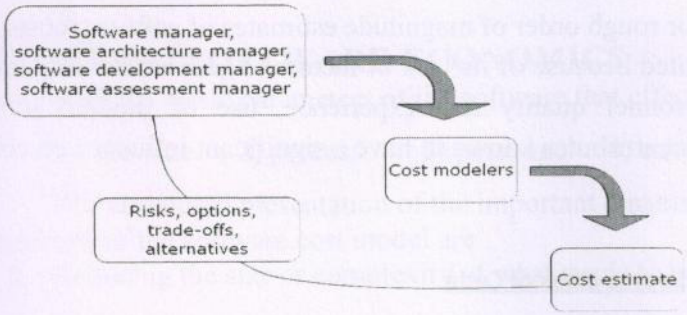


FIGURE 2-2. *Return on investment in different domains*

2.2 PRAGMATIC SOFTWARE COST ESTIMATION

The predominant cost estimation process



Debate: Which Model or Tool?

- Which cost estimation model to use
- Whether to measure software size in source line of code or function points

What constitutes a good estimate

A good estimate has the following attributes:

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

Cost Estimation Models: COCOMO

- Barry Boehm, 1981
- Royce, "HUGE BENEFIT"
- Well-documented cost estimation models
- Three Modes

- Five Basic Phase Life Cycle
- Evolution: COCOMO II

  Basic COCOMO is good for rough order of magnitude estimates of software costs, but its accuracy is necessarily limited because of its lack of factors to account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and other project attributes known to have a significant influence on costs."

Measuring Software Size:

Source lines of code versus funct Source Lines of Code

- **Barry Boehm**
- **Easy to Automate and Instrument.**
- **More ambiguous measure due to language advances and other components**
- **More useful and precise measurement basis of various metrics perspectives**

ion points

Function Points

- **Capers Jones**
- **Independent of Technology**
- **More accurate estimator in early phases**

UNIT-II

Improving Software Economics: Reducing Software Product Size, Improving software Processes, Improving Team Effectiveness, Improving Automation, Achieving Required Quality, Peer Inspections.

The old way and new way: Principles of Conventional Software Engineering, Principles of Modern Software Management, Transitioning to an Iterative Process.

IMPROVING SOFTWARE ECONOMICS

There are FIVE basic parameters of the software that effect the cost of software

1) size 2) process 3) personnel 4) environments 5) quality

The structured presentation of the important dimensions around the five basic parameters of the software cost model are

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

These parameters are given in priority order for most software domains

The following table lists some of the technology developments ,process improvement efforts , and management approaches targeted at improving software economics development and integration.

COST MODEL PARAMETERS	TRENDS
Size Abstraction and component-based development technologies	Higher order languages(C++, da 95, Java, VB..etc) Object-oriented(analysis, design, programming) Reuse Commercial components
Process Methods and techniques	Iterative development Process maturity models Architecture-first development Acquisition reform
Personal People factors	Training and personnel skill development Teamwork Win-win culture
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 Statistical quality control.

3.1 REDUCING SOFTWARE PRODUCT SIZE

The most significant way to improve affordability and return investment is usually to produce a product that achieve the design goals with the minimum amount of human-generated source material. Component-based development is introduced here as the general term for reducing the "source" language size necessary to achieve a software solution. Reuse, object-oriented technology, automatic code production, and higher order programming languages.

3.1.1 Languages

Universal function points (UFPs) are useful estimators for language-independent, early life-cycle estimates. The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries. SLOC metrics are useful estimators for software after a candidate solution is formulated and an implementation language is known. Some of these results are shown in the following table.

The data in the table illustrate why people are interested in modern languages such as C++, Ada 95, Java, and Visual Basic: The level of expressability is very attractive. However, care must be taken in applying these data because of numerous possible.

Language expressiveness of some of today's popular languages

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

Universal function points can be used to indicate the relative program sizes required to implement a given functionality. For example, to achieve a given application

with a fixed number of function points, 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++

3.1.2 OBJECT-ORIENTED METHODS AND VISUAL MODELING

Object-oriented technology is in reducing the overall size of what needs to be developed. Booch has described the following three reasons

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

Here is an example of how object-oriented technology permits corresponding improvements in teamwork and interpersonal communications.

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

This aspect of object-oriented technology enables an architecture-first process, in which integration is an early and continuous life-cycle activity.

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

This feature of object-oriented technology is crucial to the supporting languages and environments available to implement object-oriented architectures.

Booch also summarized FIVE characteristics of a successful OO project.

1. A ruthless focus on the development of a system that provides a well understood collection of essential minimal characteristics
2. The existence of a culture that is centered on results, encourages communication, and yet is not afraid to fail
3. The effective use of object-oriented modeling

4. The existence of a strong architectural vision
5. The application of a well-managed iterative and incremental development life cycle

These characteristics have little to do with object orientation. However, object oriented methods, notations, and visual modeling provide strong technology support for the process framework.

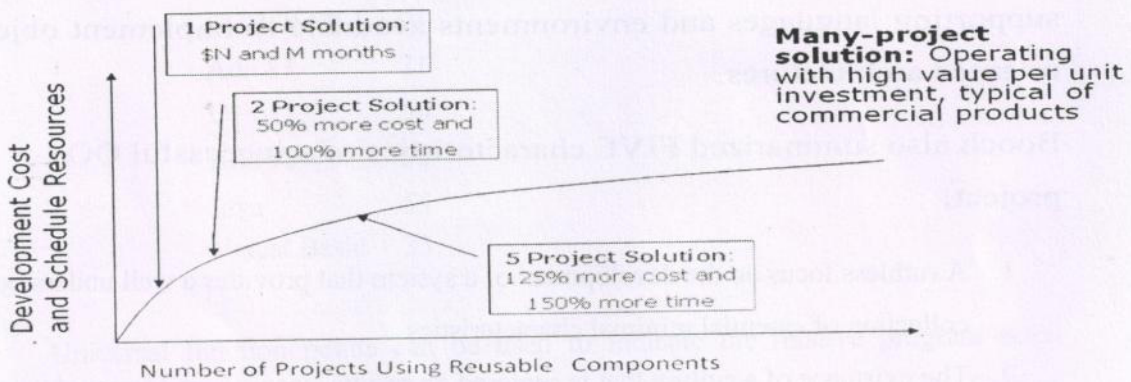
3.1.3 REUSE

Reusing existing components and building reusable components have been natural software engineering activities. Software design methods have always dealt implicitly with reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality. In general, things get reused for economic reasons.

They lack economic motivation, trustworthiness, and accountability for quality, support, improvement, and usability. Most truly reusable components of value are transitioned to commercial products supported by organizations with the following characteristics:

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

Reducing Software Product Size – Reuse



3.1.4 COMMERCIAL COMPONENTS

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

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

The selection of commercial components over development of custom components as significant impact on objects overall architecture, because the trade off's frequently have global effects on quality and cost.

3.2 IMPROVING SOFTWARE PROCESSES

Process is an overloaded term. For software-oriented organizations, there are many processes and sub processes. There are three distinct process perspectives.

- Metaprocess: It is an organization's policies, procedures, and practices for pursuing a software-intensive line of business. The focus of this process is on organizational economics, long-term strategies, and software ROI.

- Macroprocess: It is a project's policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality constraints. The focus of the macro process is on creating an adequate instance of the meta process for a specific set of constraints
- Microprocess: It is a project team's policies, procedures, and practices for achieving an artifact of the software process. The focus of the microprocess is on achieving an intermediate product baseline with adequate quality and adequate functionality as economically and rapidly as practical

Three levels of process and their attributes

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

The primary focus of process improvement should be on achieving an adequate solution in the minimum number of iterations and eliminating as much downstream scrap and rework as possible. Every instance of rework introduces a sequential set of tasks that most be redone.

3.3 IMPROVING TEAM EFFECTIVENESS

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

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

In examining how to assign staff to a software project, Boehm offered the following FIVE staffing principles

1. **The principle of top talent**: Use better and fewer people .
2. **The principle of job matching:** Fit the tasks to the skills and motivation of the people available.
3. **The principle of career progression:** An organization does best in the long run by helping its people to self-actualize.
4. **The principle of team balance:** Select people who will complement and harmonize with one another.

   The principle of team balance include the following dimensions include:

   - **Raw skills** : Intelligence objectivity, creativity, organization, analytical thinking
   - **Psychological makeup**: leaders and followers, risk takers and conservatives, visionaries and nitpickers, cynics and optimists
   - **Objectives**: financial, feature set, quality, timeliness

5. **The principle of phase-out:** Keeping a misfit on the team doesn't benefit anyone.

This is really a sub principle of the other four. A misfit gives you a reason to find a better person or to live with fewer people. A misfit demodulates other team members, will not self-actualize, and disrupts the team balance in some dimension.

The following are some crucial attributes of successful software project managers that deserve much more attention:

1. **Hiring skills**. Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.
2. **Customer-interface skill**. Avoiding adversarial relationships among stakeholders is a prerequisite for success.
3. **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.
4. **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.
5. **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.

3.4 IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

The environment has great effect on the productivity of the process. No of tools available to provide automation support for creating and evolving software engineering artifacts. These include *Planning tools, requirements management tools, visual modeling tools, compilers, editors, debuggers, quality assurance analysis tools, test tools, and user interfaces.*

An environment that provides semantic integration and process automation can improve productivity, quality and accelerate the adoption of modern techniques.

A common threat in successful S/W projects is that they hire good people and provide them with good tools to accomplish their jobs.

3.5 ACHIEVING REQUIRED QUALITY

Many of what are accepted today as software best practices are derived from the development process and technologies summarized. The following table summarizes some dimensions of quality improvement.

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
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, straight- forward 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	Over constrained	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

Key practices that improve overall software quality include the following:

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

Improved insight into run-time performance issues is even more important as projects incorporate mixtures of commercial components and custom-developed components. Conventional development processes stressed early sizing and timing

estimates of computer program resource utilization. However, the typical chronology of events in performance assessment was as follows:

- **Project inception**: The proposed design was asserted to be low risk with adequate performance margin.
- **Initial design review**: Optimistic assessments of adequate design margin were based mostly on paper analysis or rough simulation of the critical threads. In most cases, the actual application algorithms and database sizes were fairly well understood. However, the infrastructure-including the operating system overhead, the database management overhead, and the inter process 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 infra-structure was usually the scapegoat, but the real culprit was immature use of the infrastructure, immature architectural solutions, or poorly understood early design trade-offs.

3.6 PEER INSPECTIONS: A PRAGMATIC VIEW

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

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

and requirements compliance

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

Architectural issues are exposed only through more rigorous engineering activities such as the following:

- 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 m the context of critical subsets of the use cases and scenarios
- Incorporating lessons learned back into the models, use cases, implementations, and plans

Achieving architectural quality is inherent in an iterative process that evolves the artifact sets together in balance. The checkpoints along the way are numerous, including human review and inspections focused on critical issues. But these inspections are not the primary checkpoints.

UNIT III

The Old Way and The new

Over the past two decades there has been a significant re-engineering of the software development process. Many of the conventional management and technical practices have been replaced by new approaches that combine recurring themes of successful project experience with advances in software engineering technology.

4.1 THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

There are many descriptions of engineering software "the old way." After years of software development experience, the software industry has learned many lessons and formulated many principles. The following describes one view of today's software engineering principles as a bench mark for introducing the primary themes. The benchmark is a brief article titled "**Fifteen Principles of Software Engineering**". The article was subsequently expanded into a book that enumerates 201 principles. Despite its title, the articles describes the top 30 principles, and it is as good a summary as any of the conventional wisdom within the software industry.

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

development method is one that uses the same notation throughout the life cycle. Why should software engineers use Ada for requirements, design, and code unless Ada were optimal for all these phases?

8. ***Minimize intellectual distance***. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure.
9. ***Put techniques before tools***. An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer.
10. ***Get it right before you make it faster.*** It is far easier to make a working program run faster than it is to make a fast program work. Don't worry above optimization during initial coding.
11. ***Inspect code***. Inspecting the detailed design and code is a much better way to find errors than testing.
12. ***Good management is more important than good technology***. The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Good management motivates people to do their best, but there are no universal "right" styles of management.
13. ***People are the key to success***. Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process will succeed. The wrong people with appropriate tools, languages, and process will probably fail.
14. ***Follow with care***. Just because everybody is doing something does not make it right for you. It may be right, but you must carefully asses its applicability to you environment. Objects orientation, measurement, reuse, process improvement, CASE, prototyping- all these might increase quality, decrease cost, and increase user satisfaction. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal.
15. ***Take responsibility***. When a bridge collapses we ask "What did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant 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 you provide a user, the more functionality 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***. IO have often heard software engineer 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 simple, proven concept that results in software that is easier to test and much easier to maintain.

24. ***User coupling an 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 and Tom McCabe's
26. ***Don't test your own software***. Software developers should never be the primary testers of their own software.
27. ***Analyze causes for errors***. It is far more cost-effective to reduce the effect of an error by preventing it than it is to find and fix it. One way to do this is to analyze the causes of errors as they are detected.
28. ***Realize that software's entropy increases***. Any software systems 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.

4.2 THE PRINCIPLES OF MODERN SOFTWARE MANAGEMENT

Davis's format top 10 principles of modern software management. The principles are in priority order.

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 decision, and the lift-cycle plans before the resources are committed for full-scale development.
2. **Establish an iterative life-cycle process** that confronts risk early. With today's sophisticated software systems, it is not possible to define the entire problem, design the entire solution, build the software, then test the end product in sequence. Instead, an iterative process that refines the problem understanding, an effective solution, and an effective plan over several iterations encourages a balanced treatment of all stakeholder objectives. Major risks must be addressed early to increase predictability and avoid expensive downstream scarp and rework.
3. **Transition design methods to emphasize component-based development**. Moving from a line-of-code mentality to a component-based mentality is necessary to reduce the amount of human-generated source code and custom

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

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

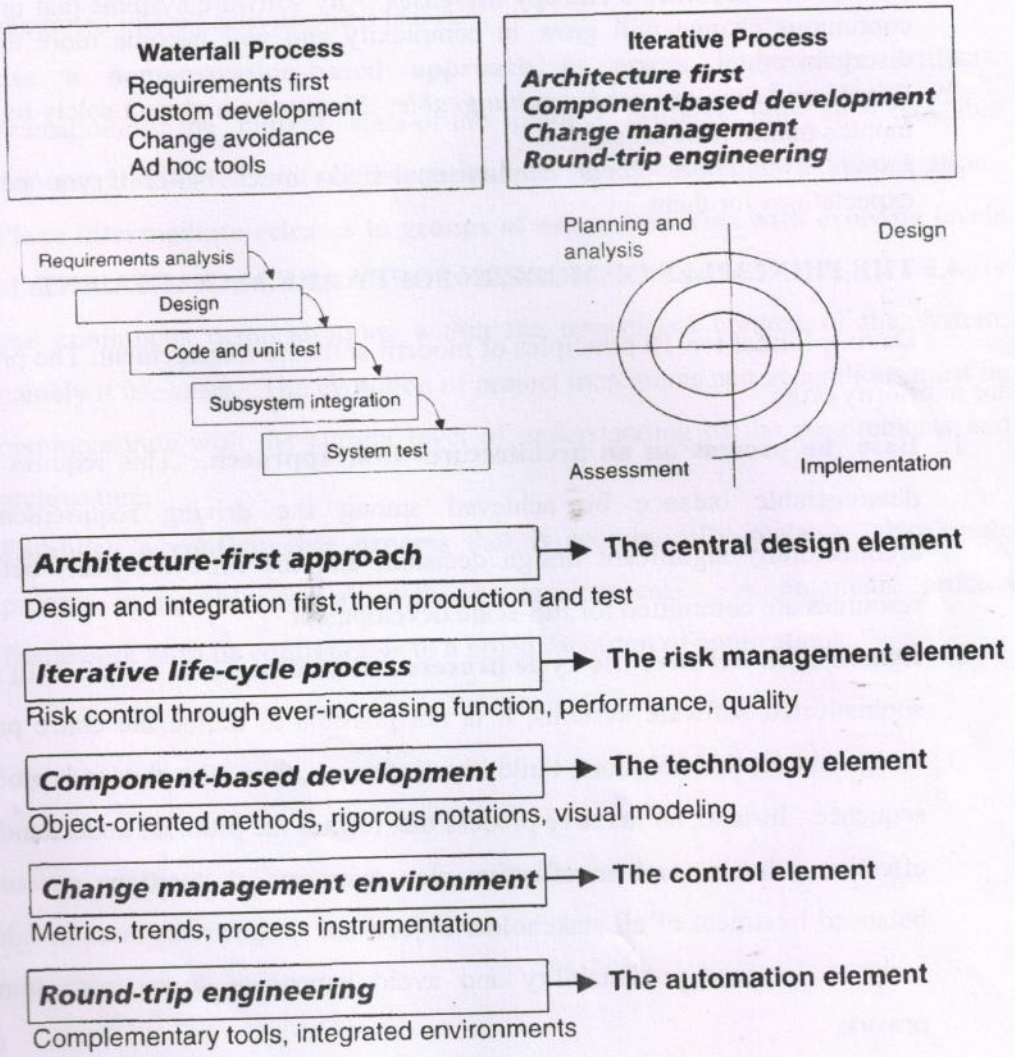


FIGURE 4-1. *The top five principles of a modern process*

5. **Enhance change freedom through tools that support round-trip engineering.** Round-trip engineering is the environment support necessary to automate and synchronize engineering information in different formats.

6. **Capture design artifacts in rigorous, model-based notation**. A model based approached 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. The best assessment mechanisms are well-defined measures derived directly from the evolving engineering artifacts and integrated into all activities and teams.
8. **Use a demonstration-based approach** to assess intermediate artifacts. Transitioning the current state-of-the product artifacts into an executable demonstration of relevant scenarios stimulates earlier convergence on integration.
9. **Plane intermediate releases in groups of usage scenarios with evolving levels of details**. It is essential that the software management process drive toward early and continuous demonstrations within the operational context of the system, namely it use cases. The evolution of project increments and generations must be commensurate with the current level of understanding of the requirements and architecture.
10. **Establish a configurable process** that is economically scalable. No single process is suitable for all software developments. A pragmatic process framework must be configurable to a broad spectrum of applications.

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

4.3 TRANSITIONING TO AN ITERATIVE PROCESS

The process exponent parameters mapping of COCOMO II model to top 10 principles of a modern process

1. **Application precedentedness.** Domain experience is a critical factor in understanding how to plan and execute a software development project. For unprecedented systems, one of the key goals is to confront risks and establish early precedents, even if they are incomplete or experimental. This is one of the primary reasons that the software industry has moved to an ***iterative lift-cycle process***. Early iterations in the life cycle establish precedents from which the product, the process, and the plans can be elaborated in ***evolving levels of detail***.
2. **Process flexibility**. Development of modern software in characterized by such a broad solution space and so many interrelated concerns that there is a paramount need for continuous incorporation of changes. These changes may be inherent in

the problem understanding, the solutions space, or the plans. Project artifacts must be supported by efficient **change management** commensurate with project needs. Both a grid process and a chaotically changing process are destined for failure except with the most trivial projects. A **configurable process** that allows a common framework to be adapted across a range of projects is necessary to achieve a software return on investment.

3. **Architecture risk resolution**. **Architecture-first** development is a crucial theme underlying a successful iterative development process. A project team develops and stabilizes an architecture before developing all the components that make up the entire suite of applications components. An **architecture-first and component-based development approach** forces the infrastructure, common mechanisms, and control mechanisms to be elaborated early in the life cycle and drives all component make/buy decisions into the architecture process. This approach initiates integration activity early in the life cycle as the verification activity of the design process and products. It also forces the development environment for life-cycle software engineering to be configured and exercised early in the life cycle, thereby ensuring early attention to testability and a foundation for **demonstration-based assessment**.
4. Team cohesion, successful teams are cohesive, and cohesive teams are successful. Successful teams and cohesive teams share common objectives and priorities. Cohesive teams avoid sources of project turbulence and entropy that may result from difficulties in synchronizing project stakeholder expectations. While there are many reasons for such turbulence, on of the primary reasons is miscommunication, particularly in exchanging information solely through paper documents that present engineering information subjectively. Advances in technology have enabled more rigorous and understandable notations for communicating software engineering information, particularly in based completely on paper exchange. These **model-based** formats have also enabled the **round-trip engineering** support needed to establish change freedom sufficient for evolving design representations.
5. **Software process maturity**. The software engineering Institute's capability Maturity Model(CMM) is well-accepted benchmark for software process

assessment. Just as domain experience is crucial for avoiding the application risks and exploiting the available domain assets and lessons learned, risks and exploiting the available domain assets and lessons learned software process maturity is crucial for avoiding software development risks and exploiting the organization's software assets and lessons learned.

Life Cycle Phases: Engineering and Production Stages, Inception. Elaboration, Construction, Transition Phases.
Artifacts Of The Process : The Artifact Sets. Management Artifacts, Engineering Artifacts, Programmatic Artifacts.

5.1 ENGINEERING AND PRODUCTION STAGES

Any software development life cycle consists of TWO stages

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 development activities

The following tables shows the 2 stages of the life cycle

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 Engineering stage composed of TWO distinct phases : INCEPTION and ELABORATION.

The production is decomposed into TWO distinct phases : CONSTRUCTION and TRANSITION. The following figure shows phases of the life cycle

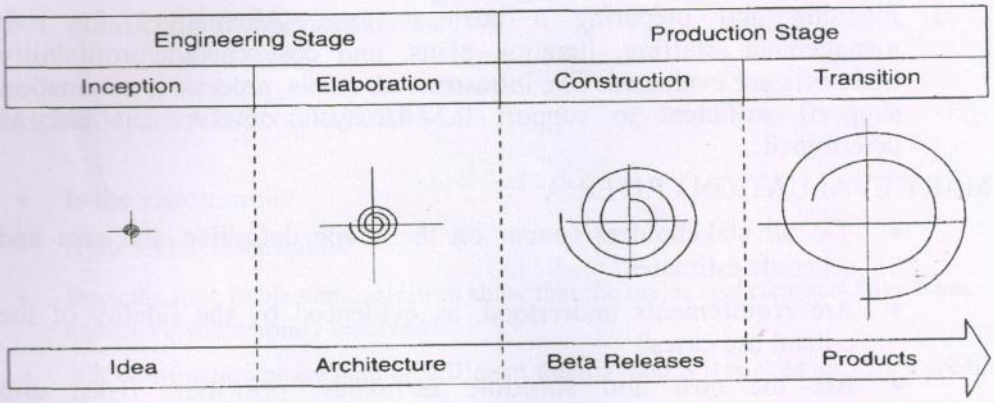


FIGURE 5-1. *The phases of the life-cycle process*

5.2 INCEPTION PHASE :

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

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES

1. Formulating the scope of the project. This activity involves capturing the requirements and operational concept in an information repository that describes the user's view of the requirements. The information repository should be sufficient to define the problem space and derive the acceptance criteria for the end product.
2. Synthesizing the architecture. Design trade-offs, problem space ambiguities, and available solution-space assets (technologies and existing components) are evaluated. An information repository is created that is sufficient to demonstrate the feasibility of at least one candidate architecture and an initial baseline of make/buy decisions so that the cost,

schedule, and resource estimates can be derived.

3. Planning and preparing a business case. Alternatives for risk management, staffing, iteration plans, and cost/schedule/profitability trade-offs are evaluated. The infrastructure (tools, processes, automation support) sufficient to support the life-cycle development task is determined.

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?
- Are actual resource expenditures versus planned expenditures acceptable?

5.3 ELABORATION PHASE

The Elaboration phase is the most critical of the FOUR Phases. During the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size risk, and novelty of the project.

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. This activity involves establishing a high-fidelity understanding of the critical use cases that drive architectural or planning decisions.
- Elaborating the process and infrastructure. The construction process, the tools and process automation support, and the intermediate milestones and their respective evaluation criteria are established.
- Elaborating the architecture and selecting components. Potential components are evaluated and make/buy decisions are sufficiently understood so that construction phase cost and schedule can be determined with confi-

dence. The selected architectural components are integrated and assessed against the primary scenarios.

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?

5.4 CONSTRUCTION PHASE

In this 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 managing resources and controlling operations to optimize costs, schedule, 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?
- Is this product baseline stable enough to be deployed in the user community?

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

5.5 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 focuses on the activities required to place the software into the hands of the users. Typically, this phase includes several iterations, including beta releases, general availability releases, and bug-fix and enhancement releases.

PRIMARY OBJECTIVES

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

ESSENTIAL ACTIVITIES

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

EVALUATION CRITERIA

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

Each of the four phases consists of one or more iterations in which some

technical capability is produced in demonstrable form and assessed against a set of criteria. An iteration represents a sequence of activities for which there is a well-defined intermediate event, the scope and results of the iteration are captured via discrete artifacts. Whereas major milestones at the end of each phase use formal versions of evaluation criteria and release descriptions, minor milestones use informal (internally controlled) versions of these artifacts. Each phase corresponds to the completion of a sufficient number of iterations to achieve a given overall project state. The transition from one phase to the next maps more to a significant business decision than to the completion of a specific software development activity.

UNIT – IV

Software architecture is the central design problem of a complex software system. An architecture is the software system design. The ultimate goal of the engineering stage is to converge on a stable architecture baseline. architecture baseline is not a paper document; it is a collection of information across all the engineering sets. A *model* is a relatively independent abstraction of a system. A *view* is a subset of a model that abstracts a specific, relevant perspective.

7.1 ARCHITECTURE: A MANAGEMENT PERSPECTIVE

The most critical technical product of a software project is its architecture : The infrastructure, control, control, and data interfaces that permit software components, to cooperate as a system and software designers to cooperate efficiently as a team.

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

1. An architecture (the intangible design concept) is the design of a software system, as opposed to the design of a component. This includes all engineering necessary to specify a complete bill of materials. Significant make/ buy decisions are resolved, and all custom components are elaborated so that individual component costs and construction/assembly costs can be determined with confidence.

2. An *architecture baseline* (the tangible artifacts) is a slice of information across the engineering artifact sets sufficient to satisfy all stakeholders that the vision (function and quality) can be achieved within the parameters of the business case (cost, profit, time, technology, 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). It includes the additional ad-hoc notation (text and graphics) necessary to clarify the information in the models. The architecture description communicates how the intangible concept is realized in the tangible artifacts.

These definitions are necessarily abstract, because architecture takes on different forms across different system domains. In particular, 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. This life-cycle event represents a transition from the engineering stage of a project, characterized by discovery and resolution of numerous unknowns, to the production stage,

characterized by management to a predictable development plan.

- 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-pay-off 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.

7.2 ARCHITECTURE: A TECHNICAL PERSPECTIVE

Software architecture encompasses the structure of software systems (the selection of elements and the composition of elements into progressively larger subsystems), their behavior (collaborations among elements), and the patterns that guide these elements, their collaborations, and their composition.

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

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 view is probably necessary in every system; the other three views can be added to deal with the complexity of the system at hand.

The following Figure summarizes the artifacts of the design set, including the architecture views and architecture description

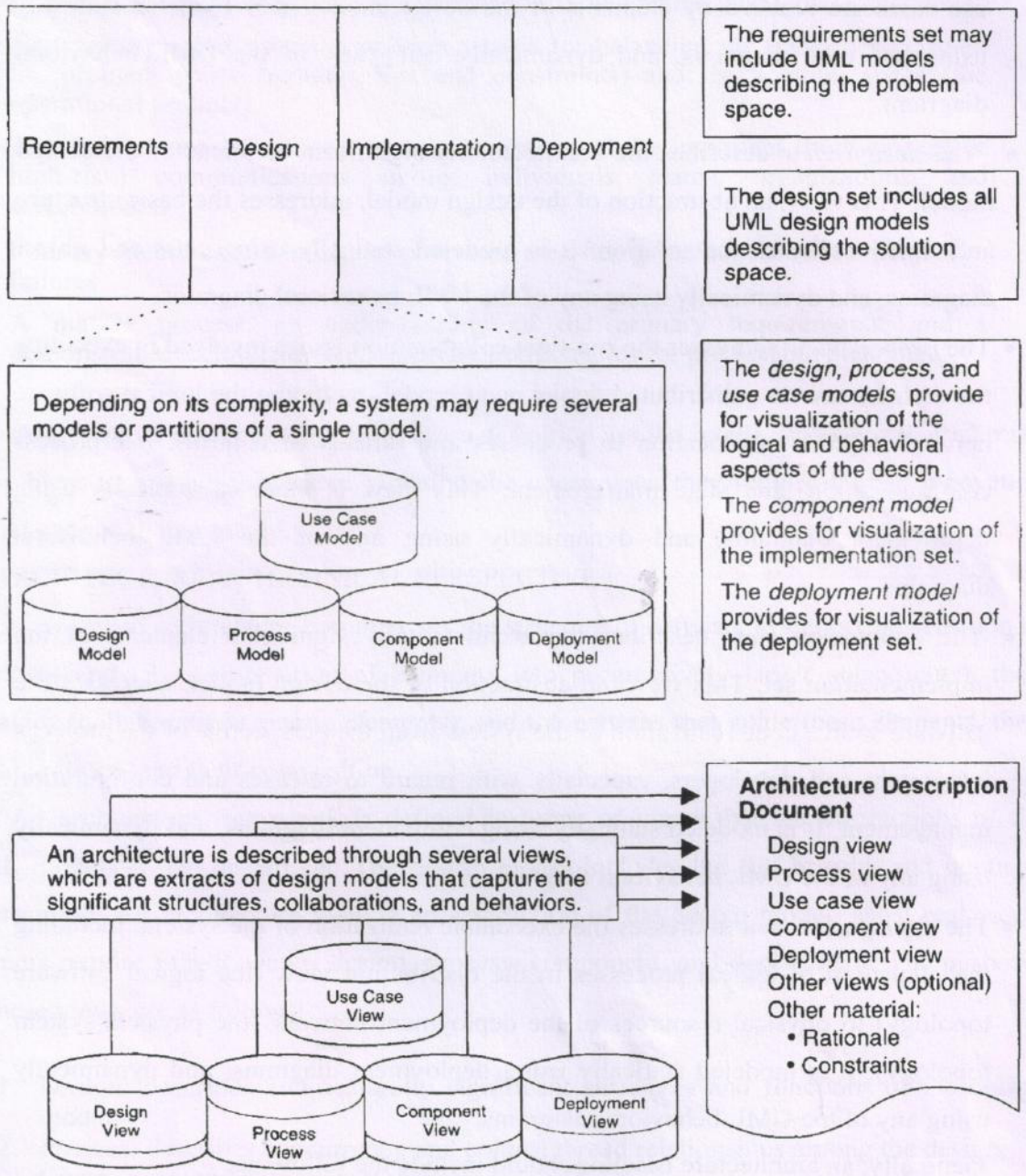


FIGURE 7-1. *Architecture, an organized and abstracted view into the design models*

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:

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

WORK FLOWS OF THE PROCESS

The activities of the process are organized into SEVEN major workflows.

1. Management workflow
2. Environment workflow
3. Requirements workflow
4 Design workflow
5. Implementation workflow
6. Assessment workflow
7. Deployment workflow

These activities are performed concurrently, with varying levels of a effort and emphasis as a project progresses through the life cycle.

8.1 SOFTWARE PROCESS WORKFLOWS

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

1. Management workflow: controlling the process and ensuring win conditions for all stakeholders
2. Environmentworkflow:automating the process&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.

The following figure illustrates the relative levels of effort expected across the phases in each of the top-level workflows. It represents one of the key signatures of a modern process framework and provides a viewpoint from which to discuss of the following key principles

1. **Architecture-first approach**: Extensive requirements analysis, design, implementation, and assessment activities are performed before the construction phase, when full-scale implementation is the focus. This early life-cycle focus on implementing and testing the architecture must precede full-scale development and testing of all the components and must precede the downstream focus on completeness and quality of the entire breadth of the product features.
2. **Iterative life-cycle process**. In the below Figure, each phase portrays at least two iterations of each workflow. This default is intended to be descriptive, not prescriptive. Some projects may require only one iteration in a phase; others may

require several iterations. The point is that the activities and artifacts of any given workflow may require more than one pass to achieve adequate results.

3. **Round-trip engineering**. Raising the environment activities to a first-class workflow is critical. The environment is the tangible embodiment of the project's process, methods, and notations for producing the artifacts.
4. **Demonstration-based approach**. Implementation and assessment activities are initiated early in the life cycle, reflecting the emphasis on constructing executable subsets of the evolving architecture.

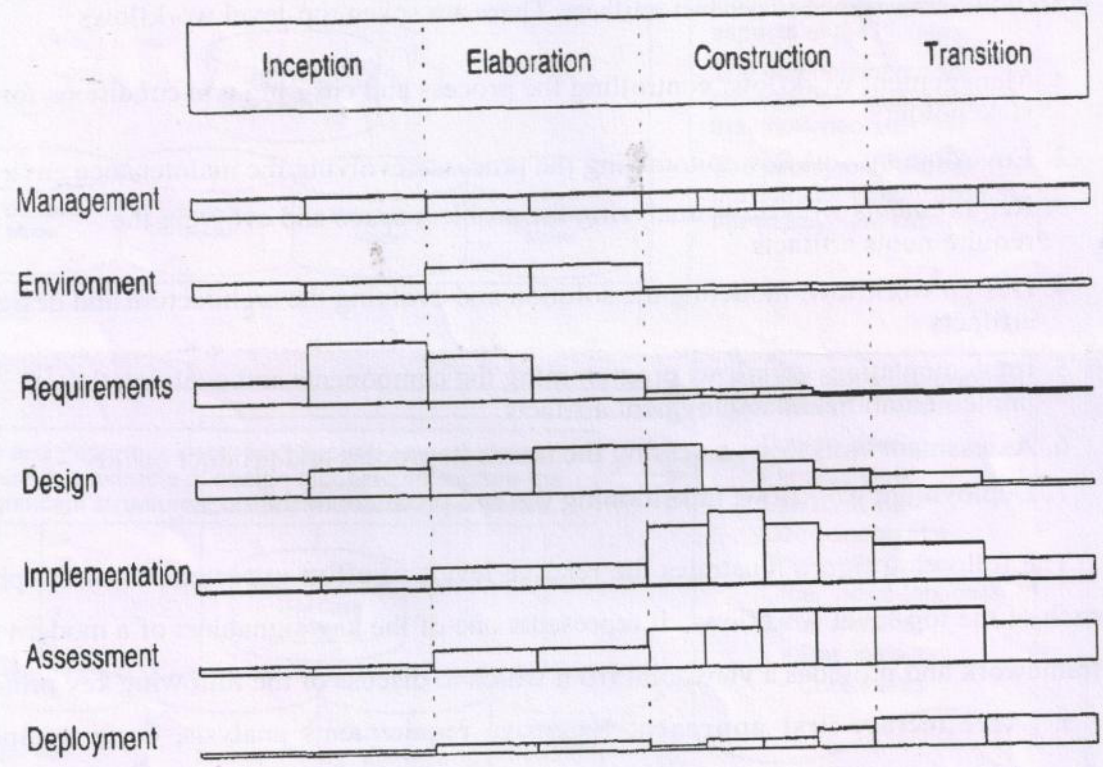


FIGURE 8-1. *Activity levels across the life-cycle phases*

The following table 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

8.2 ITERATION WORKFLOWS

An 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 located usage scenarios. The components needed to implement all selected scenarios are developed and integrated with the results of previous iterations. An individual iteration's workflow, illustrated in following figure, generally includes the following sequence

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

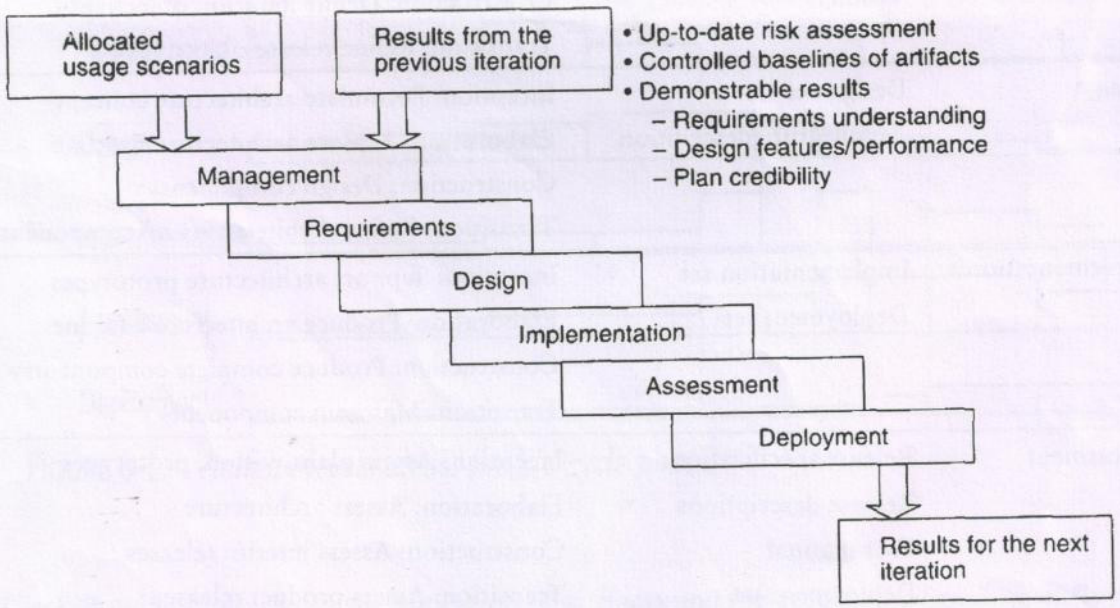
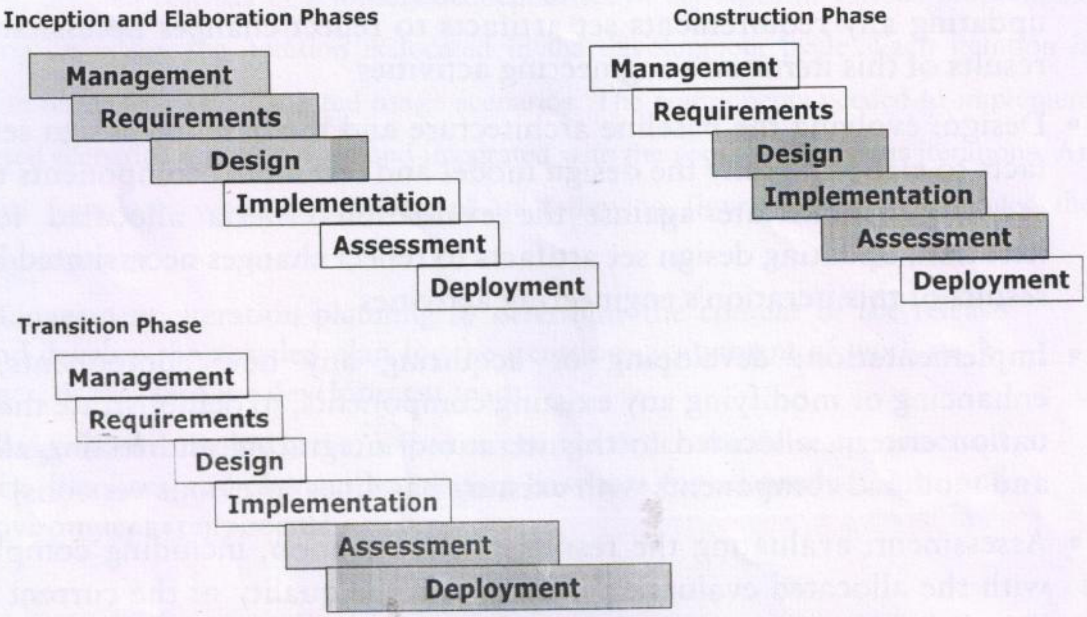


FIGURE 8-2. *The workflow of an iteration*

- Requirements: analyzing the baseline plan, the baseline architecture, and the baseline requirements set artifacts to fully elaborate the use cases to be demonstrated at the end of this iteration and their evaluation criteria; updating any requirements set artifacts to reflect changes necessitated by results of this iteration's engineering activities
- Design: evolving the baseline architecture and the baseline design set artifacts to elaborate fully the design model and test model components necessary to demonstrate against the evaluation criteria allocated to this iteration; updating design set artifacts to reflect changes necessitated by the results of this iteration's engineering activities
- 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. The following figure shows the emphasis on different activities across the life cycle.



Iteration emphasis across the life cycle

The terms *iteration* and *increment* deal with some of the pragmatic considerations. An **iteration** represents the state of the overall architecture and the complete deliverable system. An **increment** represents the current work in progress that will be combined with the preceding iteration to form the next iteration. The following figure, is an example of a simple development life cycle, illustrates the difference between iterations and increments. This example also illustrates a typical build sequence from the perspective of an abstract layered architecture.

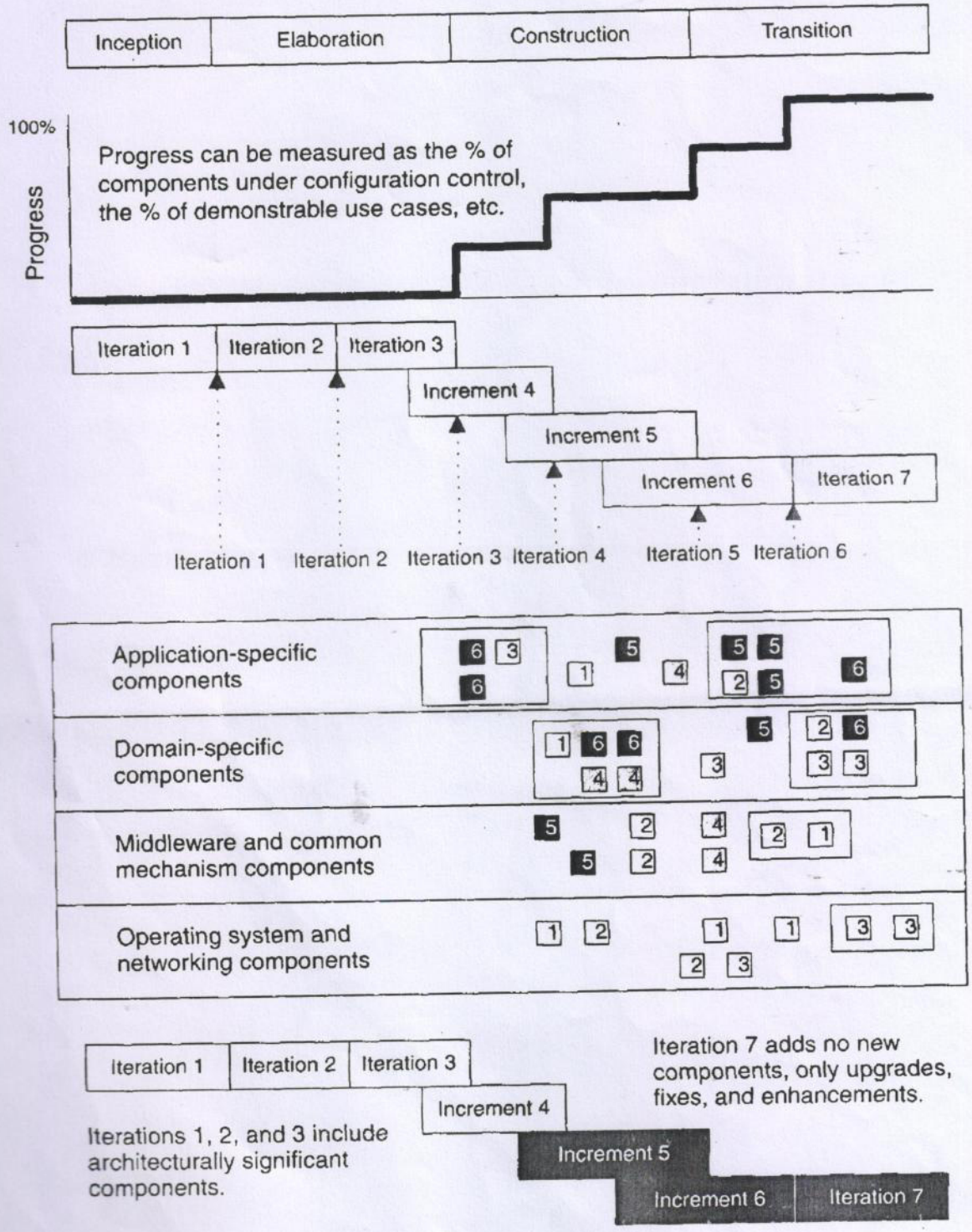


FIGURE 8-4. *A typical build sequence associated with a layered architecture*