

AD8001 SOFTWARE DEVELOPMENT PROCESSES

UNIT I SOFTWARE PROCESS 9

Software Process Maturity Software maturity Framework, Principles of Software Process Change, Software Process Assessment, The Initial Process, The Repeatable Process, The Defined Process, The Managed Process, The Optimizing Process. Process Reference Models Capability Maturity Model (CMM), CMMI, PCMM, PSP, TSP).

UNIT II SOFTWARE ECONOMICS AND LIFECYCLE 9

Software Project Management Renaissance Conventional Software Management, Evolution of Software Economics, Improving Software Economics, The old way and the new way. Life-Cycle Phases and Process artifacts Engineering and Production stages, inception phase, elaboration phase, construction phase, transition phase, artifact sets, management artifacts, engineering artifacts and pragmatic artifacts, model-based software architectures.

UNIT III SOFTWARE PROCESSES PLANNING 9

Workflows and Checkpoints of process Software process workflows, Iteration workflows, Major milestones, minor milestones, periodic status assessments. Process Planning Work breakdown structures, Planning guidelines, cost and schedule estimating process, iteration planning process, Pragmatic planning.

UNIT IV PROJECT MANAGEMENT AND METRICS

Project Organizations Line-of- business organizations, project organizations, evolution of organizations, process automation. Project Control and process instrumentation The seven-core metrics, management indicators, quality indicators, life-cycle expectations, Pragmatic software metrics, metrics automation.

UNIT V UNIT TITLE 9

CCPDS-R Case Study and Future Software Project Management Practices Modern Project Profiles, Next-Generation software Economics, Modern Process Transitions.

UNIT 2 –SOFTWARE ECONOMICS AND LIFE CYCLE

2.1 Software Project Management Renaissance

Conventional Software Management :

Conventional software management practices are sound in theory, but practice is still tied to archaic (outdated) technology and techniques.

Conventional software economics provides a benchmark of performance for conventional software management principles.

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, monitors, and control software development.

Three important analyses of the state of the software engineering industry are:

1. Software development is still highly unpredictable. Only about 10% of software projects are delivered successfully within initial budget and schedule estimates.
2. 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.
All three analyses reached the same general conclusion: The success rate for software projects is very low. The three analyses provide a good introduction to the magnitude of the software problem and the current norms for conventional software management performance.
4. the software development starts with requirements gathering phase. Then progresses through analysis, design, coding, testing and maintenance. Following figure illustrates waterfall model.

In REQUIREMENT GATHERING AND ANALYSIS phase the basic requirements of the system must be understood by software engineer, who is also called Analyst. The information domain, function, behavioral requirements of the system are understood.

All these requirements are then well documented and discussed further with the customer, for reviewing The DESIGN is an intermediate step between requirements analysis and coding. Design focuses on program attributes such as -

- Data structure
- Software architecture
- Interface representation
- Algorithmic details.

The requirements are translated in some easy to represent form using which coding can be done effectively and efficiently. The design needs to be documented for further use.

CODING is a step in which design is translated into machine-readable form. If design is done in sufficient detail then coding can be done effectively. Programs are created in this phase.

- TESTING begins when coding is done. While performing testing the major focus is on logical internals of the software. The testing ensures execution of all the paths, functional behaviors. The purpose of testing is to uncover errors, fix the bugs and meet the customer requirements.
- MAINTENANCE is the longest life cycle phase. When the system is installed and put in practical use then error may get introduced, correcting such errors and putting it in use is the major purpose of maintenance activity. Similarly, enhancing system's services as new requirements are discovered is again maintenance of the system. This model is widely used model, although it has many drawbacks. Benefits of waterfall model

The waterfall model is simple to implement.

For implementation of small systems waterfall model is useful. Drawbacks of waterfall model There are some problems that are encountered if we apply the waterfall model and those are :

- 1) It is difficult to follow the sequential flow in software development process. If some changes are made at some phases then it may cause some confusion.
- 2) The requirement analysis is done initially and sometimes it is not possible to state all the requirements explicitly in the beginning. This causes difficulty in the project.
- 3) The customer can see the working model of the project only at the end. After reviewing of the working model; if the customer gets dissatisfied then it causes serious problems.
- 4) Linear nature of waterfall model induces blocking states, because certain tasks may be dependent on some previous tasks. Hence it is necessary to accomplish all the dependent tasks first. It may cause long waiting time.

2.2 Evolution of Software Economics:

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

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

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

One important aspect of software economics (as represented within today's software cost models) is that the relationship between effort and size exhibits a diseconomy of scale. The diseconomy of scale of software development is a result of the process exponent being greater than 1.0.

Contrary to most manufacturing processes, the more software you build, the more expensive it is per unit item.

Figure 2-1 shows three generations of basic technology advancement in tools, components, and processes. The required levels of quality and personnel are assumed to be constant. The ordinate of the graph refers to software unit costs (pick your favorite: per SLOC, per function point, per component) realized by an organization.

The three generations of software development are defined as follows:

- 1) Conventional: 1960s and 1970s, craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable in that cost, schedule, and quality objectives were almost always underachieved.
- 2) Transition: 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.

Some of the components (<30%) were available as commercial products, including the operating system, database management system, networking, and graphical user interface.

- 3) Modern practices: 2000 and later, software production. This book's philosophy is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70%) off-the-shelf components. Perhaps as few as 30% of the components need to be custom built technologies for environment automation, size reduction, and process improvement are not independent of one another. In each new era, the key is complementary growth in all technologies.

For example, the process advances could not be used successfully without new component technologies and increased tool automation.

Organizations are achieving better economies of scale in successive technology eras-with very large projects (systems of systems), long-lived products, and lines of business comprising multiple similar projects. Figure 2-2 provides an overview of how a return on investment (ROI) profile can be achieved in subsequent efforts across life cycles of various domains.

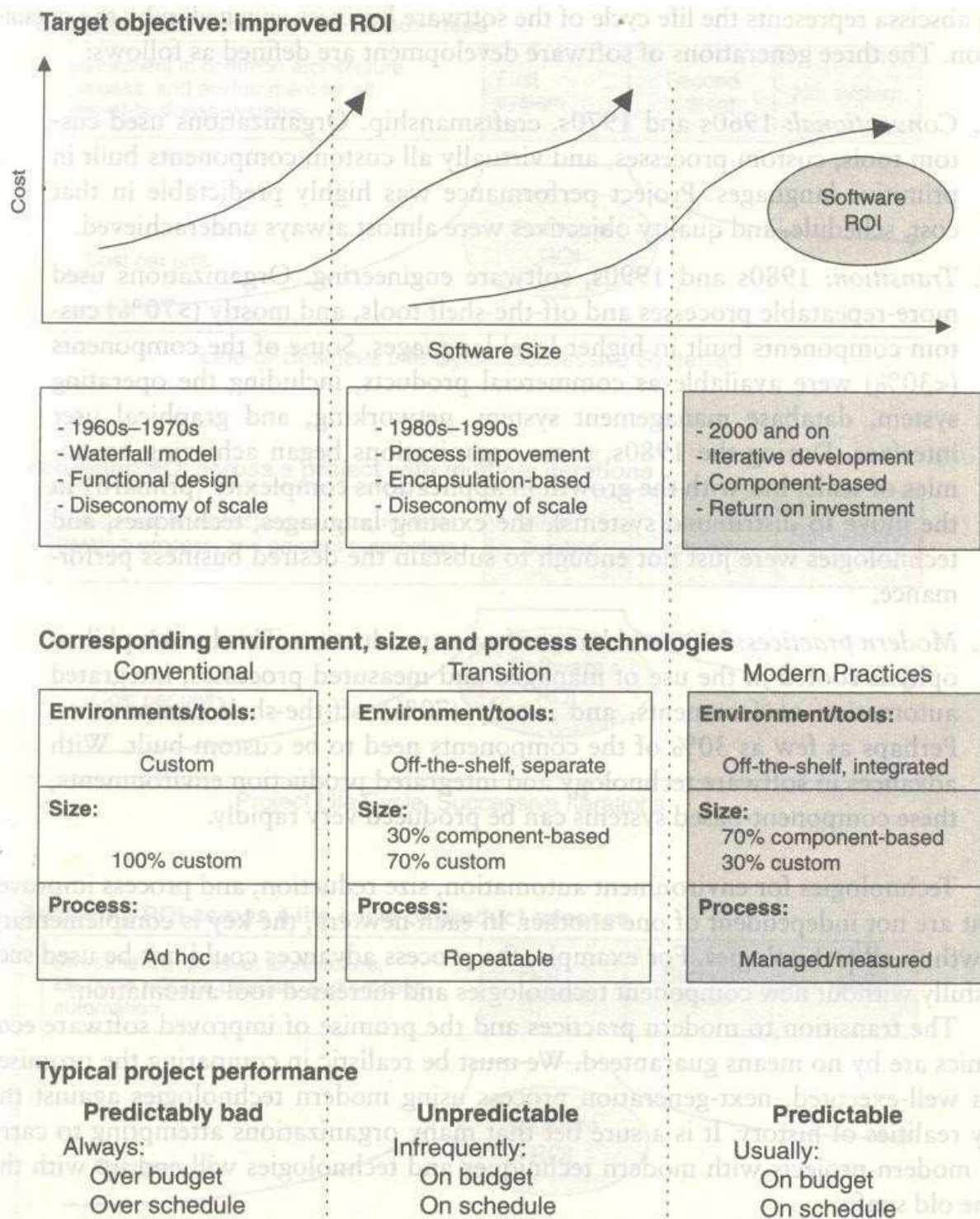


FIGURE 2-1. Three generations of software economics leading to the target objective

PRAGMATIC SOFTWARE COST ESTIMATION

One critical problem in software cost estimation is a lack of well-documented case studies of projects that used an iterative development approach. Software industry has inconsistently defined metrics or atomic units of measure, the data from actual projects are highly suspect in terms of consistency and comparability.

It is hard enough to collect a homogeneous set of project data within one organization; it is extremely difficult to homogenize data across different organizations with different processes, languages, domains, and so on.

There have been many debates among developers and vendors of software cost estimation models and tools. Three topics of these debates are of particular interest here:

1. Which cost estimation model to use?
2. Whether to measure software size in source lines of code or function points.
3. What constitutes a good estimate?

There are several popular cost estimation models (such as COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20). COCOMO is also one of the most open and well-documented cost estimation models. The general accuracy of conventional cost models (such as COCOMO) has been described as "within 20% of actuals, 70% of the time."

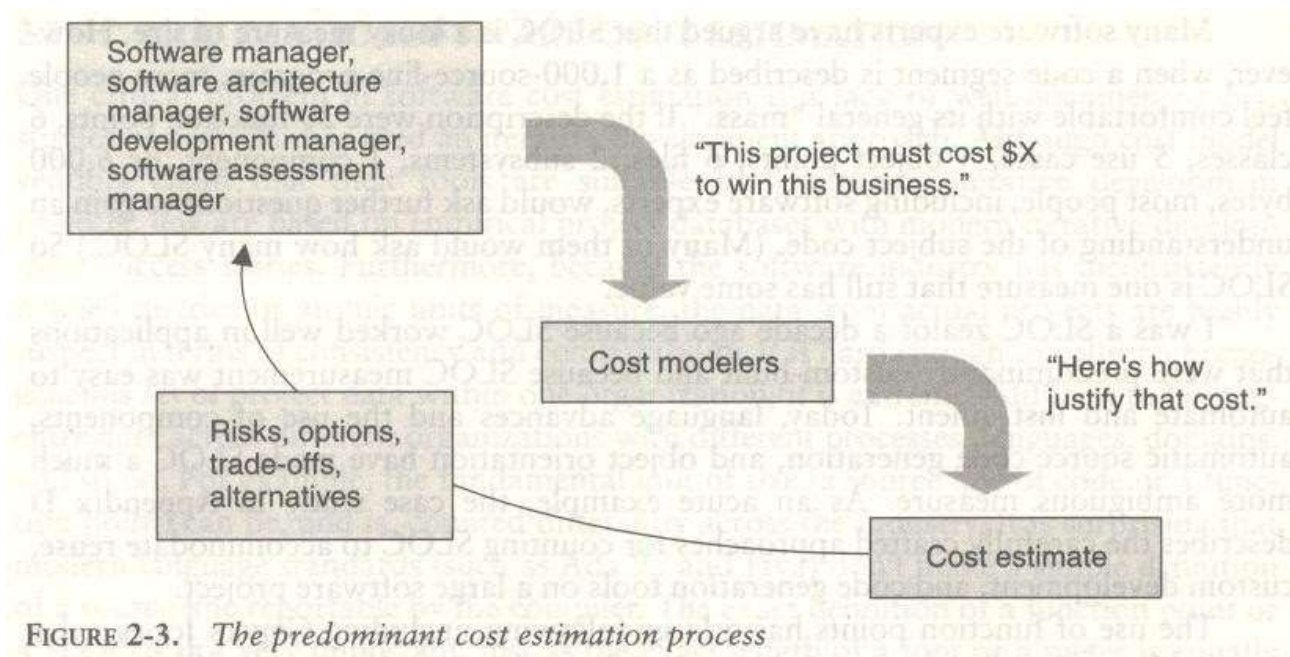
Most real-world use of cost models is bottom-up (substantiating a target cost) rather than top-down (estimating the "should" cost). Figure 2-3 illustrates the predominant practice: The software project manager defines the target cost of the software, and then manipulates the parameters and sizing until the target cost can be justified. The rationale for the target cost maybe to win a proposal, to solicit customer funding, to attain internal corporate funding, or to achieve some other goal.

The process described in Figure 2-3 is not all bad. In fact, it is absolutely necessary to analyze the cost risks and understand the sensitivities and trade-offs objectively. It forces the software project manager to examine the risks associated with achieving the target costs and to discuss this information with other stakeholders.

A good software cost 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, similar technologies, similar environments, similar quality requirements, and similar people.
- It is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

Extrapolating from a good estimate, an ideal estimate would be derived from a mature cost model with an experience base that reflects multiple similar projects done by the same team with the same mature processes and tools



3.3 Improving Software Economics

Five basic parameters of the software cost model are

- 1.Reducing the size or complexity of what needs to be developed.
- 2.Improving the development process.
- 3.Using more-skilled personnel and better teams (not necessarily the same thing).
- 4.Using better environments (tools to automate the process).

Trading off or backing off on quality thresholds

TABLE 3-1. Important trends in improving software economics

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

These parameters are given in priority order for most software domains. Table 3-1 lists some of the technology developments, process improvement efforts, and management approaches targeted at improving the economics of software development and integration.

REDUCING SOFTWARE PRODUCT SIZE

The most significant way to improve affordability and return on investment (ROI) is usually to produce a product that achieves the design goals with the minimum amount of human-generated source material. Component-based development is introduced as the general term for reducing the "source" language size to achieve a software solution.

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

size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic), automatic code generators (CASE tools, visual modeling tools, GUI builders), reuse of commercial components (operating systems, windowing environments, database management systems, middleware, networks), and object-oriented technologies (Unified Modeling Language, visual modeling tools, architecture frameworks).

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

Universal function points (UFPs¹) are useful estimators for language- independent, early life-cycle estimates. The basic units of function points are external user inputs, external outputs, internal logical data groups, external data interfaces, and external inquiries. SLOC metrics are useful estimators for software after a candidate solution is formulated and an implementation language is known.

Substantial data have been documented relating SLOC to function points. Some of these results are shown in Table 3-2. Languages expressiveness of some of today's popular languages.

TABLE 3-2. *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

OBJECT-ORIENTED METHODS AND VISUAL MODELING

Object-oriented technology is not germane to most of the software management topics discussed here, and books on object-oriented technology abound. Object-oriented programming languages appear to benefit both software productivity and software quality.

The fundamental impact of object-oriented technology is in reducing the overall size of what needs to be developed.

People like drawing pictures to explain something to others or to themselves. When they do it for software system design, they call these pictures diagrams or diagrammatic models and the very notation for them a modeling language.

These are interesting examples of the interrelationships among the dimensions of improving software economics.

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

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

1. A ruthless focus on the development of a system that provides a well understood collection of essential minimal characteristics.
2. The existence of a culture that is 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.

REUSE

Reusing existing components and building reusable components have been natural software engineering activities since the earliest improvements in programming languages.

With reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality.

Try to treat reuse as a mundane part of achieving a return on investment

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

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

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

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

COMMERCIAL COMPONENTS

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

Table 3-3 identifies some of the advantages and disadvantages of using commercial components.

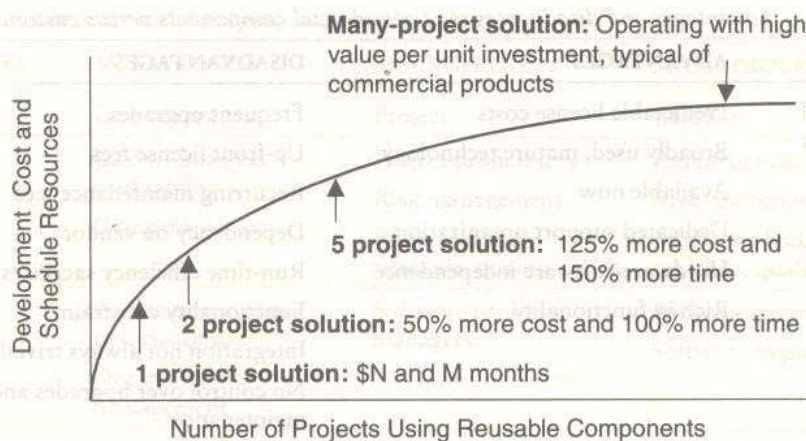


FIGURE 3-1. Cost and schedule investments necessary to achieve reusable components

TABLE 3-3. Advantages and disadvantages of commercial components versus custom software

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

IMPROVING SOFTWARE PROCESSES

Process is an overloaded term. Three distinct process perspectives are.

- **Metaprocess:** an organization's policies, procedures, and practices for pursuing a software-intensive line of business. The focus of this process is on organizational economics, long-term strategies, and software ROI.
- **Macroprocess:** a project's policies, procedures, and practices for producing a complete software product within certain cost, schedule, and quality constraints. The focus of the macro process is on creating an adequate instance of the Meta process for a specific set of constraints.
- **Microprocess:** a project team's policies, procedures, and practices for achieving an artifact of the software process. The focus of the micro process is on achieving an intermediate product baseline with adequate quality and adequate functionality as economically and rapidly as practical.
- Although these three levels of process overlap somewhat, they have different objectives, audiences, metrics, concerns, and time scales as shown in Table 3-4

In a perfect software engineering world with an immaculate problem description, an obvious solution space, a development team of experienced geniuses, adequate resources, and stakeholders with common goals, we could execute a software development process in one iteration with almost no scrap and rework.

Because we work in an imperfect world, however, we need to manage engineering activities so that scrap and rework profiles do not have an impact on the win conditions of any stakeholder. This should be the underlying premise for most process improvements.

TABLE 3-4. *Three levels of process and their attributes*

ATTRIBUTES	METAPROCESS	MACROPROCESS	MICROPROCESS
Subject	Line of business	Project	Iteration
Objectives	Line-of-business profitability	Project profitability	Resource management
	Competitiveness	Risk management	Risk resolution
		Project budget, schedule, quality	Milestone budget, schedule, quality
Audience	Acquisition authorities, customers	Software project managers	Subproject managers
	Organizational management	Software engineers	Software engineers
Metrics	Project predictability	On budget, on schedule	On budget, on schedule
	Revenue, market share	Major milestone success	Major milestone progress
		Project scrap and rework	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 EFFECTIVENESS

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

- A well-managed project can succeed with a nominal engineering team.
 - A mismanaged project will almost never succeed, even with an expert team of engineers.
 - A well-architected system can be built by a nominal team of software builders.
 - A poorly architected system will flounder even with an expert team of builders.
- Boehm five staffing principles are
1. The principle of top talent: Use better and fewer people
 2. The principle of job matching: Fit the tasks to the skills and motivation of the people available.
 3. The principle of career progression: An organization does best in the long run by helping its people to self-actualize.
 4. The principle of team balance: Select people who will complement and harmonize with one another
 5. The principle of phase-out: Keeping a misfit on the team doesn't benefit anyone
- Software project managers need many leadership qualities in order to enhance team effectiveness. The following are some crucial attributes of successful software project managers that deserve much more attention:

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.

Decision-making skill. The jillion books written about management have failed to provide a clear definition of this attribute. We all know a good leader when we run into one, and decision-making skill seems obvious despite its intangible definition.

Team-building skill. Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.

Selling skill. Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives. In practice, selling requires continuous negotiation, compromise, and empathy

IMPROVING AUTOMATION THROUGH SOFTWARE ENVIRONMENTS

The tools and environment used in the software process generally have a linear effect on the productivity of the process.

Planning tools, requirements management tools, visual modeling tools, compilers, editors, debuggers, quality assurance analysis tools, test tools, and user interfaces provide crucial automation support for evolving the software engineering artifacts. Above all, configuration management environments provide the foundation for executing and instrument the process. At first order, the isolated impact of tools and automation generally allows improvements of 20% to 40% in effort.

However, tools and environments must be viewed as the primary delivery vehicle for process automation and improvement, so their impact can be much higher. Automation of the design process provides payback in quality, the ability to estimate costs and schedules, and overall productivity using a smaller team.

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

Forward engineering is the automation of one engineering artifact from another, more abstract representation. For example, compilers and linkers have provided automated transition of source code into executable code.

Reverse engineering is the generation or modification of a more abstract representation from an existing artifact (for example, creating a .visual design model from a source code representation). Economic improvements associated with tools and environments.

It is common for tool vendors to make relatively accurate individual assessments of life-cycle activities to support claims about the potential economic impact of their tools. For example, it is easy to find statements such as the following from companies in a particular tool.

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

ACHIEVING REQUIRED QUALITY

Software best practices are derived from the development process and technologies. Table 3-5 summarizes some dimensions of quality improvement.

Key practices that improve overall software quality include the following:

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

Using metrics and indicators to measure the progress and quality of an architecture as it evolves from a high-level prototype into a fully compliant product
 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

- o Project inception. The proposed design was asserted to be low risk with adequate performance margin.
- o 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.
- o 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.
- o Integration and test. Serious performance problems were uncovered, necessitating fundamental changes in the architecture. The underlying infrastructure was usually the scapegoat, but the real culprit was immature use of the infrastructure, immature architectural solutions, or poorly understood early design trade-offs.

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

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

PEER INSPECTIONS: A PRAGMATIC VIEW

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

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

engineering artifacts

- Major milestone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant use cases
- Environment tools (compilers, debuggers, analyzers, automated test suites) that ensure representation rigor, consistency, completeness, and change control
- Life-cycle testing for detailed insight into critical trade-offs, acceptance criteria, and requirements compliance

Change management metrics for objective insight into multiple-perspective change trends and convergence or divergence from quality and progress goals
Inspections are also a good vehicle for holding authors accountable for quality products.

All authors of software and documentation should have their products scrutinized as a natural by-product of the process. Therefore, the coverage of inspections should be across all authors rather than across all components.

3.4 THE OLD WAY AND THE NEW THE PRINCIPLES OF CONVENTIONAL SOFTWARE ENGINEERING

- Make quality #1. Quality must be quantified and mechanisms put into place to motivate its achievement
- .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
- .Give products to customers early. No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it
- 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
-
- 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.
-
- 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.
- 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.
- Minimize intellectual distance. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure

- Put techniques before tools. An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer
- 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
- Inspect code. Inspecting the detailed design and code is a much better way to find errors than testing
- 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.
- People are the key to success. Highly skilled people with appropriate experience, talent, and training are key.
- 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.
- 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.
- 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.
- 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.
- 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.
- Design for change. The architectures, components, and specification techniques you use must accommodate change.
- 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."
- Use tools, but be realistic. Software tools make their users more efficient.
- 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

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

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

- 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

- Don't test your own software. Software developers should never be the primary testers of their own software.
- 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
- Realize that software's entropy increases. Any software system that undergoes continuous change will grow in complexity and will become more and more disorganized. People and time are not interchangeable. Measuring a project solely by person-months makes little sense
- 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 an iterative 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 source code and custom development.
 4. Establish a change management environment. The dynamics of iterative development, including concurrent workflows by different teams working on shared artifacts, necessitates objectively controlled baselines.
 5. Enhance change freedom through tools that support round-trip engineering. Round-trip engineering is the environment support necessary to automate and synchronize engineering information in different formats (such as requirements specifications, design models, source code, executable code, test cases).
 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 modern process

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

TRANSITIONING TO AN ITERATIVE PROCESS

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

The economic benefits inherent in transitioning from the conventional waterfall model to an iterative development process are significant but difficult to quantify. As one benchmark of the expected economic impact of process improvement, consider the process exponent parameters of the COCOMO II model. (Appendix B provides more detail on the COCOMO model) This

exponent can range from 1.01 (virtually no diseconomy of scale) to 1.26 (significant diseconomy of scale). The parameters that govern the value of the process exponent are application precedentedness, process flexibility, architecture risk resolution, team cohesion, and software process maturity.

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

- Application precedentedness. Domain experience is a critical factor in understanding how to plan and execute a software development project. For unprecedented systems, one of the key goals is to confront risks and establish early precedents, even if they are incomplete or experimental. This is one of the primary reasons that the software industry has moved to an iterative life-cycle process. Early iterations in the life cycle establish precedents from which the product, the process, and the plans can be elaborated in evolving levels of detail.
 - Process flexibility. Development of modern software is characterized by such a broad solution space and so many interrelated concerns that there is a paramount need for continuous incorporation of changes. These changes may be inherent in the problem understanding, the solution space, or the plans. Project artifacts must be supported by efficient change management commensurate with project needs. A configurable process that allows a common framework to be adapted across a range of projects is necessary to achieve a software return on investment.
 - Architecture risk resolution. Architecture-first development is a crucial theme underlying a successful iterative development process. A project team develops and stabilizes architecture before developing all the components that make up the entire suite of applications components.
 - An architecture-first and component-based development approach forces the infrastructure, common mechanisms, and control mechanisms to be elaborated early in the life cycle and drives all component make/buy decisions into the architecture process.
 - Team cohesion. Successful teams are cohesive, and cohesive teams are successful. Successful teams and cohesive teams share common objectives and priorities. Advances in technology (such as programming languages, UML, and visual modeling) have enabled more rigorous and understandable notations for communicating software engineering information, particularly in the requirements and design artifacts that previously were ad hoc and based completely on paper exchange. These model-based formats have also enabled the round-trip engineering support needed to establish change freedom sufficient for evolving design representations.
 - Software process maturity. The Software Engineering Institute's Capability Maturity Model (CMM) is a well-accepted benchmark for software process assessment. One of key themes is that truly mature processes are enabled through an integrated environment that provides the appropriate level of automation to instrument the process for objective quality control.
- ### 3.5 Life-Cycle Phases and Process artifacts
- #### Life cycle phases
-

Characteristic of a successful software development process is the well-defined separation between "research and development" activities and "production" activities. Most unsuccessful projects exhibit one of the following characteristics:

- An overemphasis on research and development
- An overemphasis on production.
Successful modern projects-and even successful projects developed under the conventional process-tend to have a very well-defined project milestone when there is a noticeable transition from a research attitude to a production attitude. Earlier phases focus on achieving

functionality. Later phases revolve around achieving a product that can be shipped to a customer, with explicit attention to robustness, performance, and finish.

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

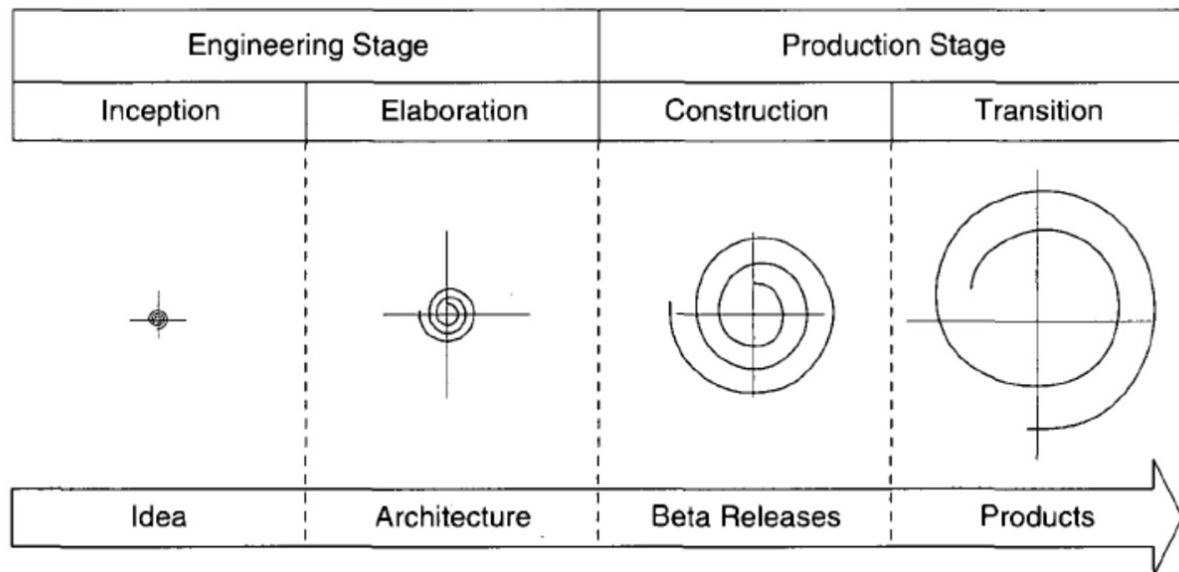
- Evolution of the plans, requirements, and architecture, together with welldefined synchronization points
- Risk management and objective measures of progress and quality
- Evolution of system capabilities through demonstrations of increasing functionality
- ENGINEERING AND PRODUCTION STAGES
- To achieve economies of scale and higher returns on investment, we must move toward a software manufacturing process driven by technological improvements in process
- automation and component- based development. Two stages of the life cycle are:
The engineering stage, driven by less predictable but smaller teams doing design and synthesis activities
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



3.7 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
- 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. 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.
- Planning and preparing a business case. Alternatives for risk

management, staffing, iteration plans, cost/schedule/profitability trade-offs are evaluated.

PRIMARY EVALUATION CRITERIA

and

- 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? (The primary value of prototyping candidate architecture is to provide a vehicle for understanding the scope and assessing the credibility of the development group in solving the particular technical problem.)
- Are actual resource expenditures versus planned expenditures acceptable

3.8 ELABORATION PHASE

At the end of this phase, the "engineering" is considered complete. The elaboration phase activities must ensure that the architecture, requirements, and plans are stable enough, and the risks sufficiently mitigated, that the cost and schedule for the completion of the development can be predicted within an acceptable range. During the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size, & risk.

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?

3. 9 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:

Beta testing to validate the new system against user expectations

Beta testing and parallel operation relative to a legacy system it is replacing Conversion of operational databases

Training of users and maintainers

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

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, field personnel training)
- Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set

EVALUATION CRITERIA

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

3.10 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

3.11 THE MANAGEMENT SET

The management set captures the artifacts associated with process planning and execution. These artifacts use ad hoc notations, including text, graphics, or whatever representation is required to capture the "contracts" among project personnel (project management, architects, developers, testers, marketers, administrators), among stakeholders (funding authority, user, software project manager, organization manager, regulatory agency), and between project personnel and stakeholders.

Specific artifacts included in this set are the work breakdown structure (activity breakdown and financial tracking mechanism), the business case (cost, schedule, profit expectations), the release specifications (scope, plan, objectives for release baselines), the software development plan (project process instance), the release descriptions (results of release baselines), the status assessments (periodic snapshots of project progress), the software change orders (descriptions of discrete baseline changes), the deployment documents (cutover plan, training course, sales rollout kit), and the environment (hardware and software tools, process automation, & documentation).

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.

Requirement 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.

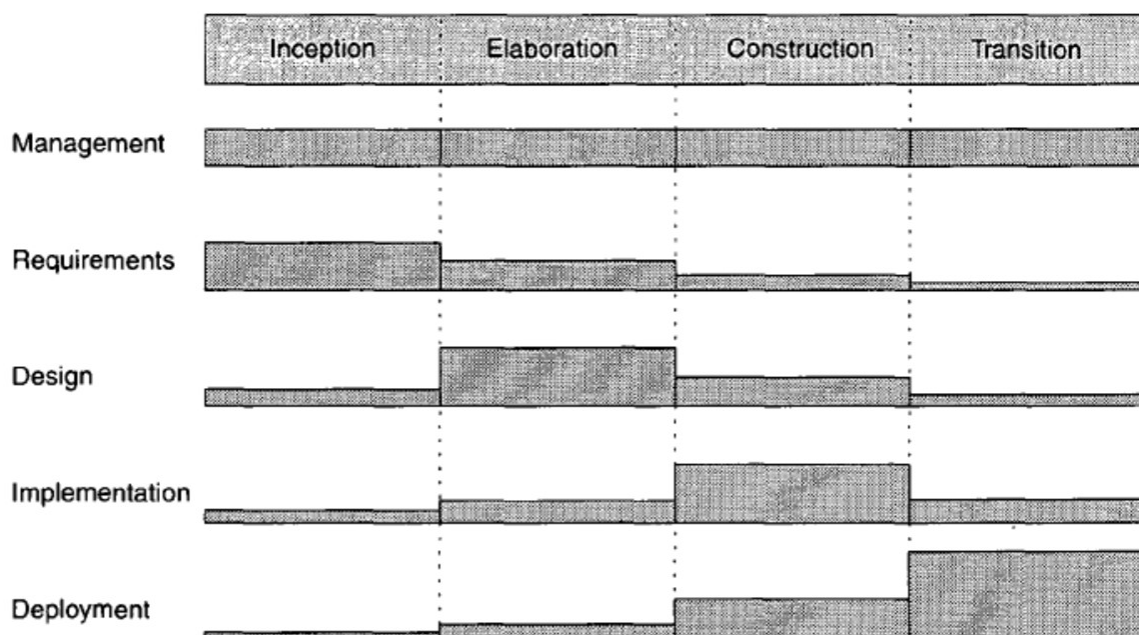


FIGURE 6-2. *Life-cycle focus on artifact sets*

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

UARTIFACT 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.

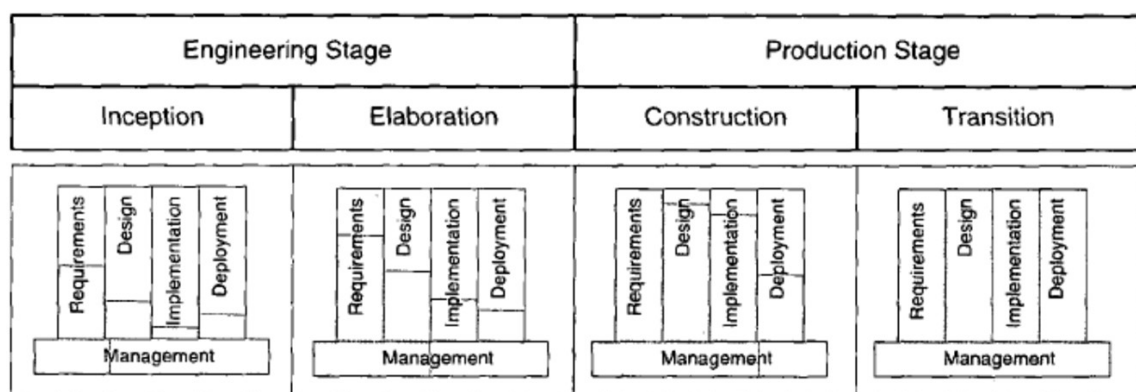


FIGURE 6-3. *Life-cycle evolution of the artifact sets*

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 non deliverable 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.

- 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 6-6. *Typical software development plan 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

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.

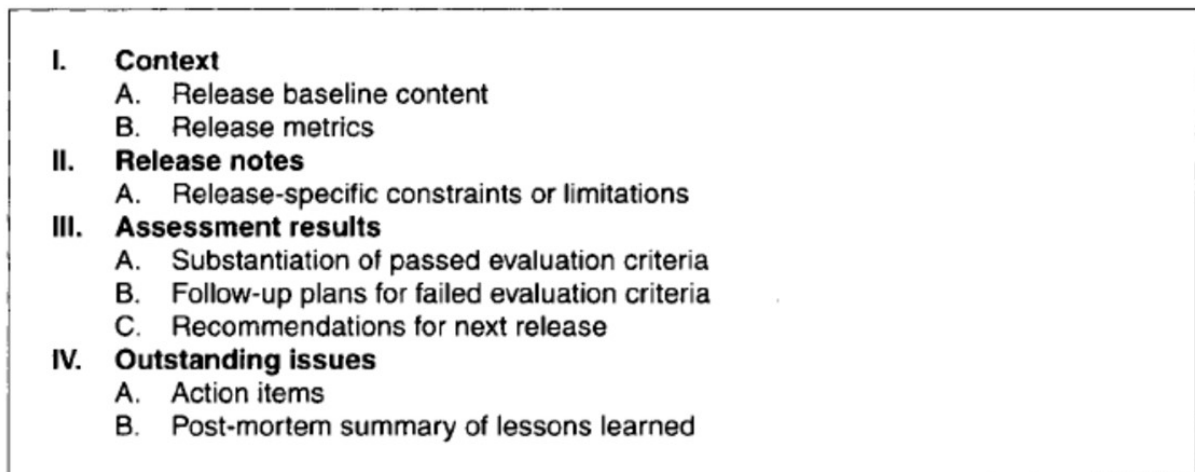


FIGURE 6-7. *Typical release description outline*

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.

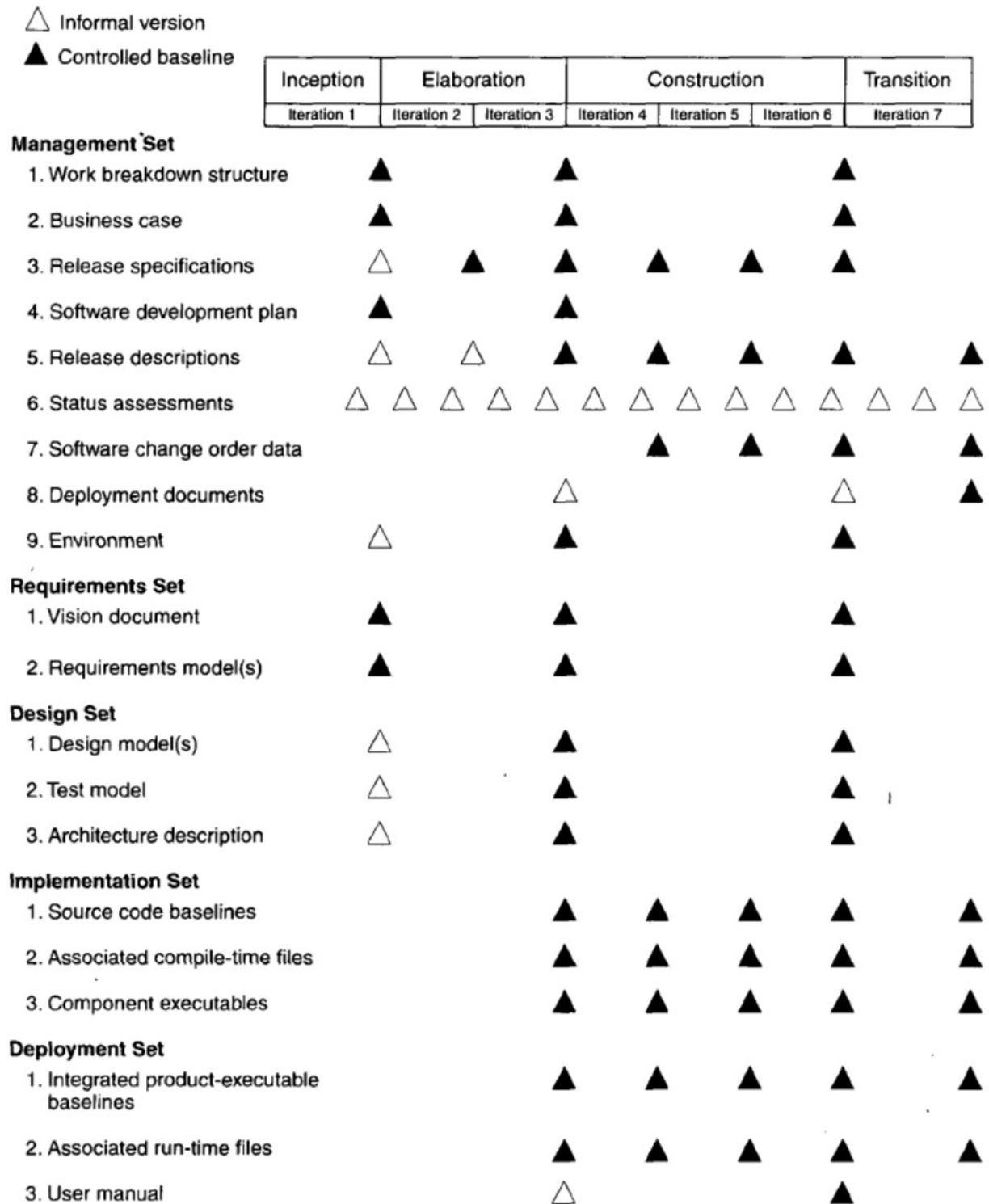


FIGURE 6-8. Artifact sequences across a typical life cycle

3.12 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

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. 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*

- 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*

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

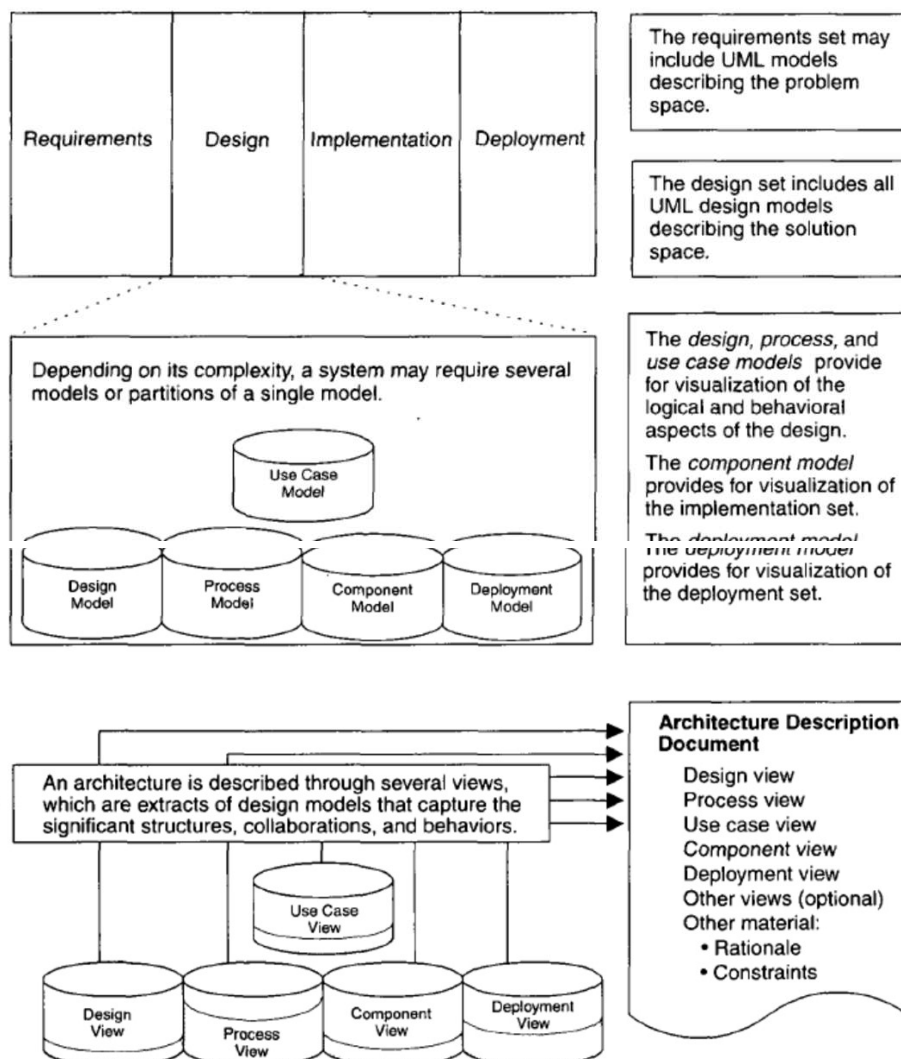


FIGURE 7-1. Architecture, an organized and abstracted view into the design models

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.

Assignments

Assignments

1. X and Y are starting a bed and breakfast service in a small town. They will have four rooms for guests. They want a system to manage the reservation and to monitor expenses and profits.

When a potential customer calls for reservation, they will check the calendar and if there is a vacancy, they will enter the customer name, address, phone number, dates, agreed upon price, credit card number and room number (s). Reservation must be guaranteed by 1 day's payment. Reservation will be held without guarantee for an agreed upon time. If not guaranteed by that date, the reservation will be dropped. Perform the following tasks :

- (a) Draw the use case, define all the classes and draw an object diagram.
- (b) Draw the sequences and collaborative diagram.

2. A magazine is published weekly and is sent by post to its subscribers. Two months before the expiry of subscription, a reminder is sent to the subscribers. If subscription is not received within a month, another reminder is sent. If renewal subscription is not received upto two weeks before the expiry of the subscription, the subscriber's name is removed from the mailing list and the subscriber is informed. Perform the following tasks :

- (a) Draw the use case, define all the classes and draw an object diagram.
- (b) Draw the sequences and collaborative diagram.

Part A – Q & A

Unit - II

PART - A Questions

1. EXPLAIN PHASES OF THE WATERFALL MODEL. (CO2,K2)

The five-main phase of waterfall model are: Requirements gathering
Design
Development
Testing
Implementation & Maintenance

2. WHAT IS CONVENTIONAL SOFTWARE MANAGEMENT? (CO2,K1)

Organizations used conventional software management. This management utilized custom tools and process and virtually custom components built-in primitive languages. Thus, the performance of the project was very much predictable in the schedule, cost, and quality.

3. WHAT IS CONVENTIONAL SOFTWARE ECONOMICS? (CO2,K1)

Conventional software economics provides a benchmark of performance for conventional software management principles.

4. WRITE TWO PRIMARY OBJECTIVES OF INCEPTION PHASE. (CO2,K1)

The primary goals of the Inception phase are to achieve stakeholder consensus regarding the objectives of the project and to obtain funding

5. WRITE TWO ESSENTIAL ACTIVITIES OF ELABORATION PHASE. (CO2,K1)

The essential activities of the elaboration phase are: Elaborating the vision establishing a solid understanding of the most critical use cases that drive the architectural and planning decisions. Elaborating the process and the infrastructure, the development environment.

6. WRITE ANY TWO PRIMARY EVALUATION CRITERIA OF CONSTRUCTION PHASE. (CO2,K1)

Elaborating the vision.

Elaborating the process and infrastructure.

Elaborating the architecture and selecting components.

7. LIST THE ACTIVITIES OF TRANSITION PHASE. (CO2,K1)

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

8. WHAT ARE THE FIVE TYPES OF ARTIFACT SETS. (CO2,K1)

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).

9. DEFINE WORK BREAKDOWN STRUCTURE. (CO2,K1)

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

10. WHAT ARE THE THREE FUNDAMENTAL FLAWS THAT SUFFER THE CONVENTIONAL WORK BREAKDOWN STRUCTURE? (CO2,K1)

Conventional work breakdown structures frequently suffer from three fundamental flaws.

1. They are prematurely structured around the product design.
2. They are prematurely decomposed, planned, and budgeted in either too much or too little detail.
3. They are project-specific, and cross-project comparisons are usually difficult or impossible.

Part B – Questions

Part-B Questions

1	Explain about conventional software management	CO2	K2
2	Explain about the evolution of software economics	CO2	K2

3	Explain about the Improving software economics.	CO2	K2
4	Differentiate conventional versus modern software project management.	CO2	K2
5	Explain about the artifacts of the process.	CO2	K2
6	Explain in detail about the workflows of the process	CO2	K2
7	Explain about inception and elaboration phase	CO2	K2
8	Explain about construction and transition phase	CO2	K2
9	Explain in detail about engineering sets	CO2	K2

Mini Project Suggestions

Mini Project

Develop suitable Software Life Cycle model for the following projects

1. Uberization Project – In-plant Vehicle movement
2. Rack monitoring
3. print a online visitor diary after retirement.
4. Online display of OEE for all Plants