UNIT - II

Software Project Management Renaissance
Conventional Software Management, Evolution of Software Economics, Improving Software Economics, The old way and the new way.
Life-Cycle Phases and 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.

2. Software Project Management Renaissance

2.1 Conventional Software Management

The waterfall model of conventional software management, which is still prevalent in many mature software organizations, has served its purpose. The ever-increasing market demands on software development performance continue. The increasing breadth of Internet applications has further accelerated the transition to a more modern management process known as spiral, incremental, evolutionary, or iterative development. A comparison of conventional and modern software development models illustrates some of the critical discriminators in this transition.
Top 10 Principles of Conventional Software Management Most software engineering texts present the waterfall model as the source of the conventional software management process. Conventional software management techniques work well for custom-developed software where the requirements are fixed when development begins. The life cycle typically follows a sequential transition from requirements to design to code to testing, with ad hoc documentation that attempts to capture complete intermediate representations at every stage. After coding and unit testing individual components, the components are compiled and linked together (integrated) into a complete system.

The top 10 principles of conventional software management are:

1. Freeze requirements before design.
2. Forbid coding before detailed design review.
3. Use a higher-order programming language.
4. Complete unit testing before integration.
5. Maintain detailed traceability among all artifacts.
6. Thoroughly document each stage of the design.
7. Assess quality with an independent team.
8. Inspect everything.
9. Plan everything early with high fidelity.
10. Rigorously control source-code baselines.

Significant inconsistencies among component interfaces and behavior, which can be extremely difficult to resolve, cannot be identified until integration, which almost always takes much longer than planned. Budget and schedule pressures drive teams to shoehorn in the quickest fixes. Redesign usually is out of the question. Testing of system threads, operational usefulness, and requirements compliance is performed through a series of releases until the software is judged adequate for the user. About 90% of the time, the process results in a late, over-budget, fragile and expensive-to-maintain software system.

A typical result of following the waterfall model is that integration and testing consume too much time and effort in major software development workflows. For successful projects, about 40% of resources go to integration and testing. The percentage is even higher for unsuccessful projects. With such a low success rate, better risk management is imperative.

2.2 Evolution Of Software Economics

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 that 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. Organiz:1tions 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.

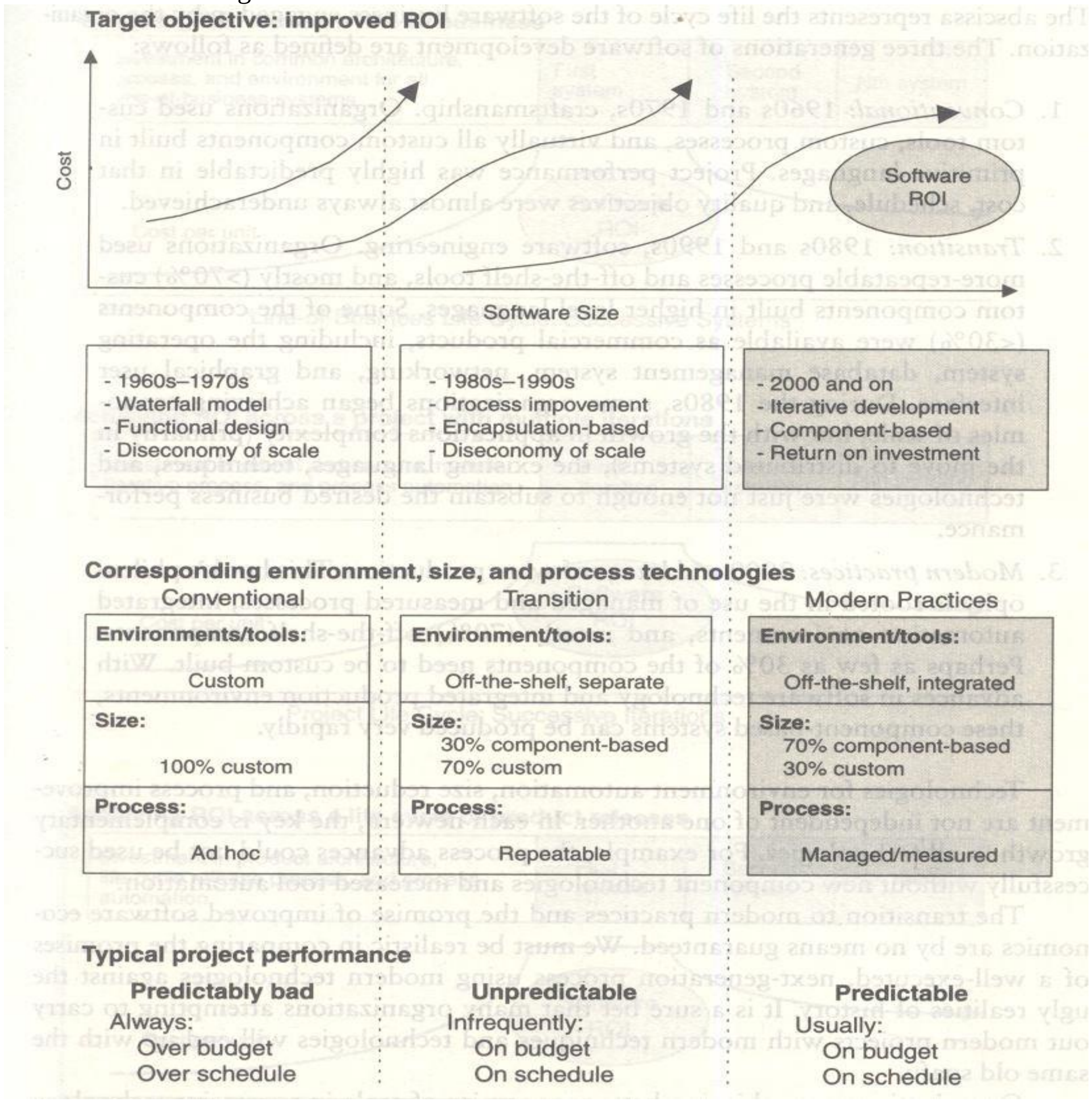
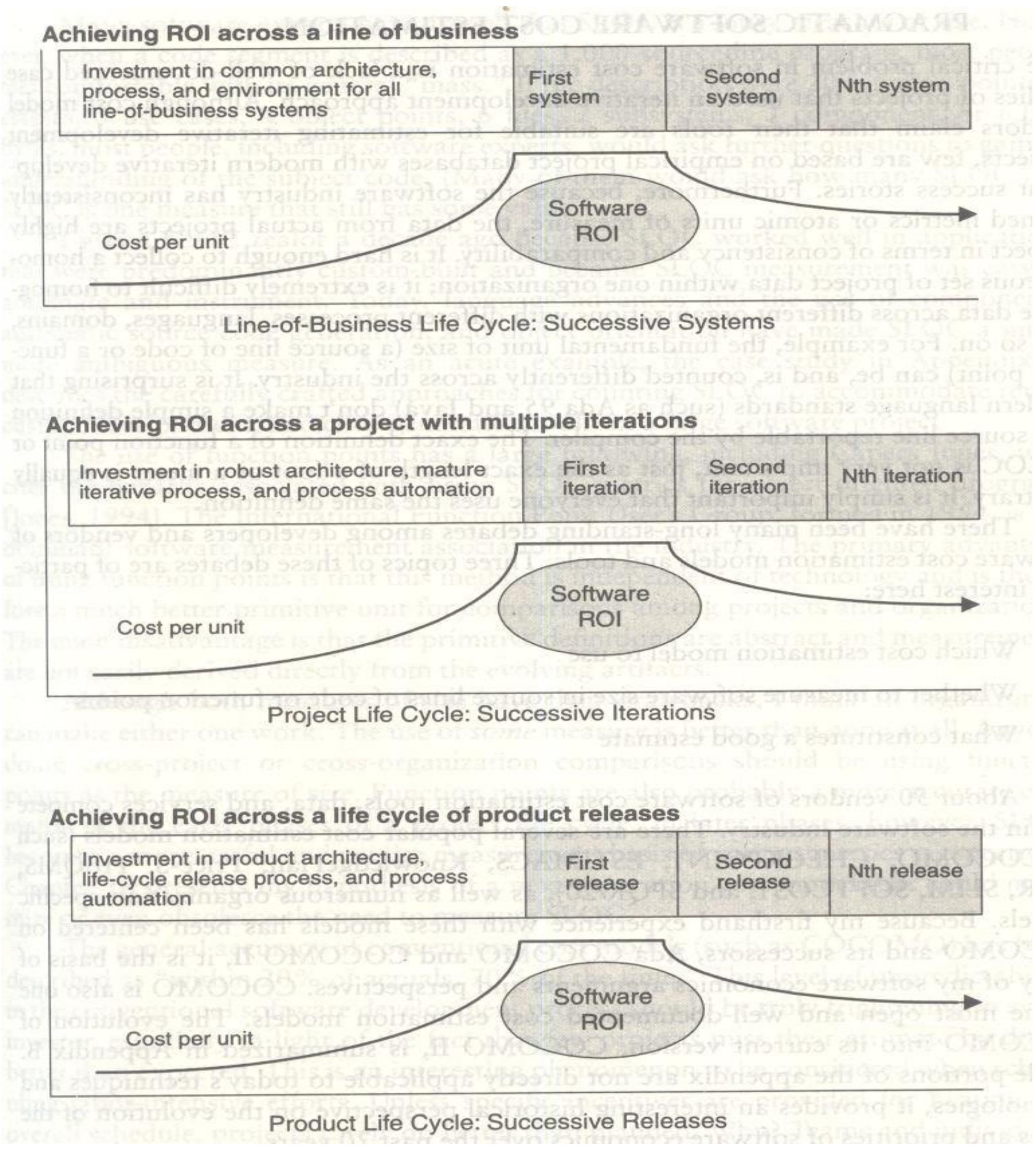


Figure 2 – 1 Three Generation of Software Economics

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.



Achieving ROI across a line of business
Investment in common architecture, process, and environment for all line-of-business systems
First system
Second system
Nth system
Software ROI
Cost per unit
Line-of-Business Life Cycle: Successive Systems
Achieving ROI across a project with multiple iterations
Investment in robust architecture, mature iterative process, and process automation
First iteration
Second iteration
Nth iteration
Software ROI
Cost per unit
Project Life Cycle: Successive Iterations
Achieving ROI across a life cycle of product releases
Investment in product architecture, life-cycle release process, and process automation
First release
Second release
Nth release
Software ROI
Cost per unit
Product Life Cycle: Successive Releases

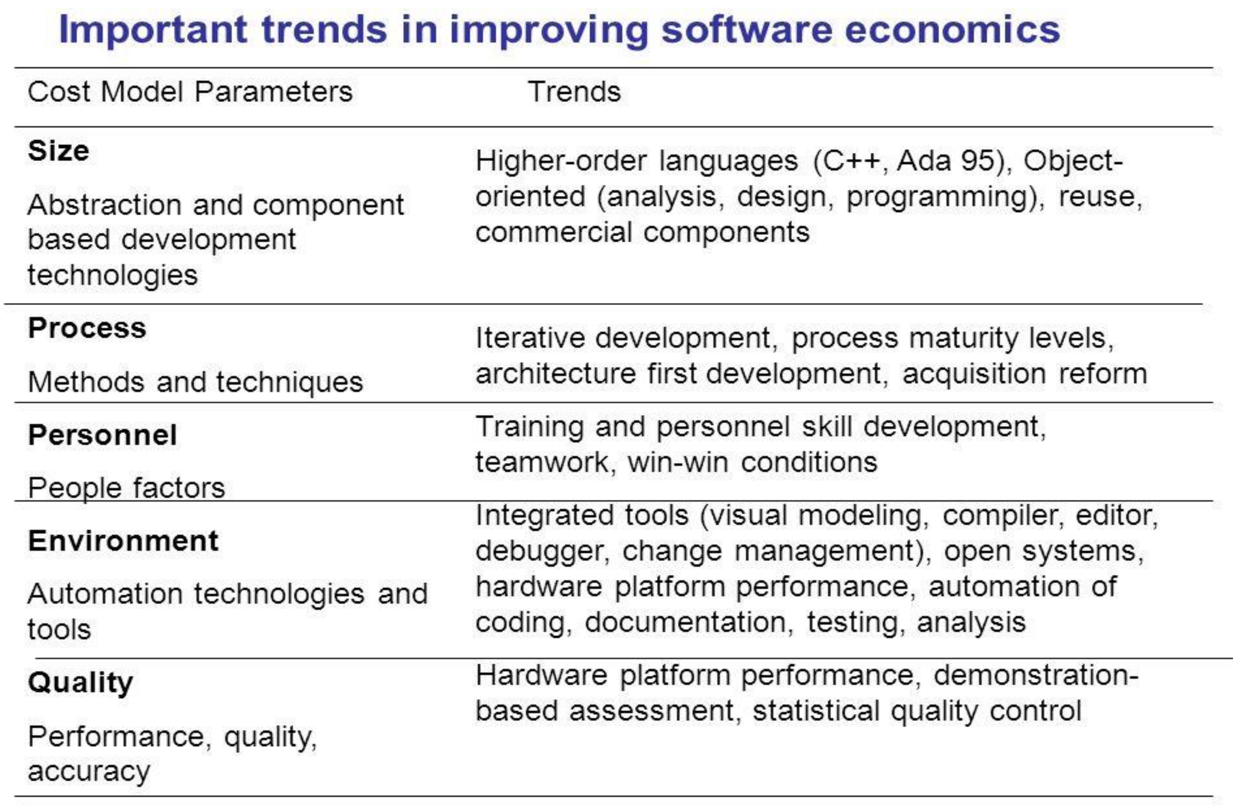
Figure 2-2: Return on Investment in different domains

2.3 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)
5. Trading off or backing off on quality thresholds

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.



Important trends in improving software economics

Cost Model Parameters	Trends
Size Abstraction and component based development technologies	Higher-order languages (C++, Ada 95), Object-oriented (analysis, design, programming), reuse, commercial components
Process Methods and techniques	Iterative development, process maturity levels, architecture first development, acquisition reform
Personnel People factors	Training and personnel skill development, teamwork, win-win conditions
Environment Automation technologies and tools	Integrated tools (visual modeling, compiler, editor, debugger, change management), open systems, hardware platform performance, automation of coding, documentation, testing, analysis
Quality Performance, quality, accuracy	Hardware platform performance, demonstration-based assessment, statistical quality control

Figure 2-3: Improving software Economics

2.4 The Old Way and The New

The Principles of Conventional Software Engineering

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

2. **High-quality software is possible**. Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people

3. **Give products to customers early**. No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it

4. **Determine the problem before writing the requirements**. When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution

6.**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.

7.**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.

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 about optimization during initial coding

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

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

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

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

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

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

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

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

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

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

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

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

23. **Encapsulate**. Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.
24. **Use coupling and cohesion**. Coupling and cohesion are the best ways to measure software's inherent maintainability and adaptability

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

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

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

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

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

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

The Principles Of Modern Software Management

Top 10 principles of modern software management are. (The first five, which are the main themes of an iterative process, are summarized in Figure 2-4.)

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.

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.

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.

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

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).
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.

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.

Use a demonstration-based approach to assess intermediate artifacts.

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.

Establish a configurable process that is economically scalable. No single process is suitable for all software developments.

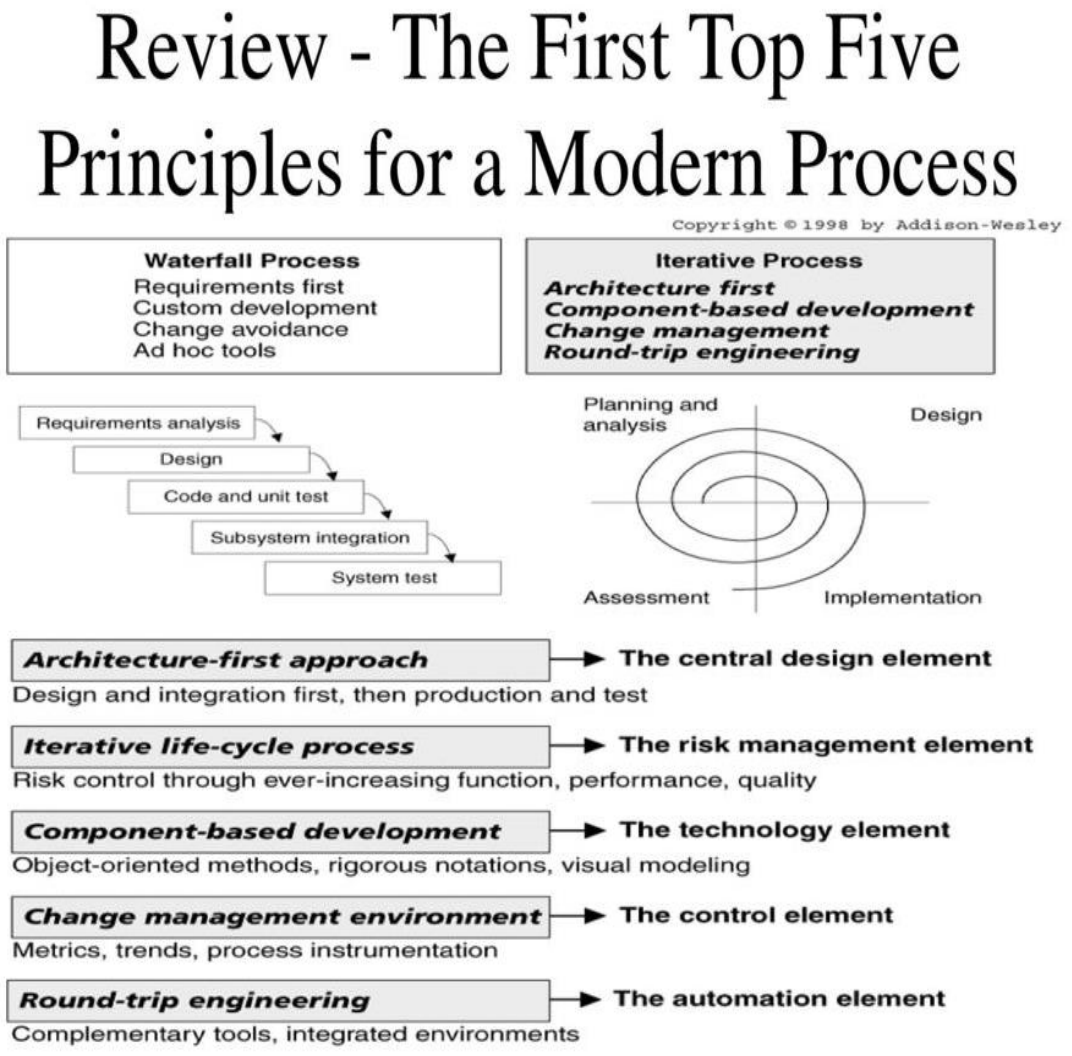

Review - The First Top Five
Principles for a Modern Process
Copyright © 1998 by Addison-Wesley
Waterfall Process
Requirements first
Custom development
Change avoidance
Ad hoc tools
Iterative Process
Architecture first
Component-based development
Change management
Round-trip engineering
Requirements analysis
Design
Code and unit test
Subsystem integration
System test
Planning and analysis
Design
Assessment
Implementation
Architecture-first approach
The central design element
Design and integration first, then production and test
Iterative life-cycle process
The risk management element
Risk control through ever-increasing function, performance, quality
Component-based development
The technology element
Object-oriented methods, rigorous notations, visual modeling
Change management environment
The control element
Metrics, trends, process instrumentation
Round-trip engineering
The automation element
Complementary tools, integrated environments

Figure 2-4: Top Five Modern Process

2.5 Life-Cycle Phases

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

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

and deployment activities

Table 2-1: Life Cycle Aspects

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

The engineering stage is decomposed into two distinct phases, inception and elaboration, and the production stage into construction and transition. These four phases of the life-cycle process are loosely mapped to the conceptual framework of the spiral model.
The size of the spiral model corresponds to the inertia of the project with respect to the breadth and depth of the artifacts that have been developed.

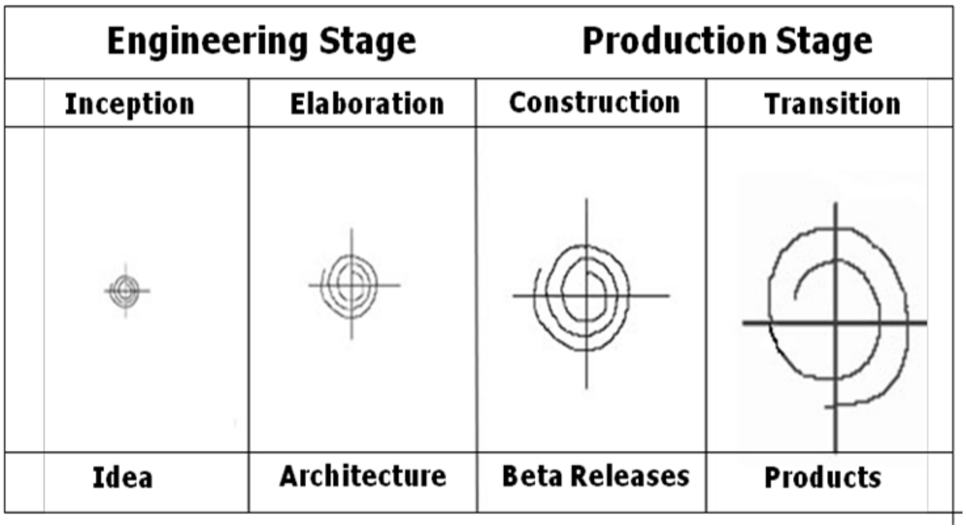


Figure 2-5: Life Cycle Stages

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

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

Inception Phase

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

Elaboration Phase

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

Essential activities:

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

Construction Phase

During the construction phase :

All remaining components and application features are integrated into the application

All features are thoroughly tested

Essential activities:

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

- **Transition Phase**

The transition phase is entered when baseline is mature enough to be deployed in the end-user domain. This phase could include beta testing, conversion of operational databases, and training of users and maintainers.

Essential activities:

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

Evaluation Criteria

Is the user satisfied?

Are actual resource expenditures versus planned expenditures acceptable?

Each of the four phases consists of one or more iterations in which some technical capability is produced in demonstrable form and assessed against a set of the criteria.

The transition from one phase to the nest maps more to a significant business decision than to the completion of specific software activity.

2.6 Process Artifacts

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

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

Management set:

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

a. Relevant stakeholder reviews

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

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

The Engineering Sets:

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

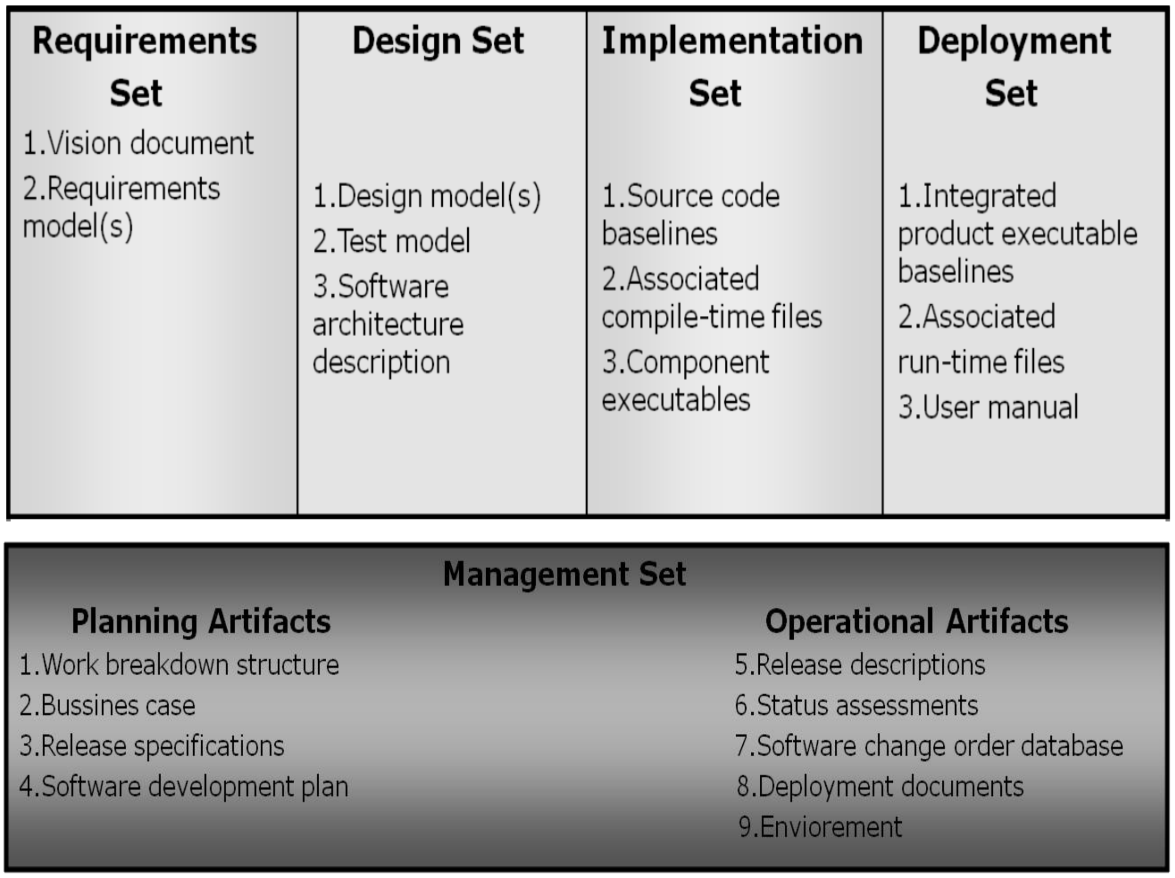


Figure 2-6: Management Set

2.7 Management artifacts

The *management* set includes several artifacts

Work Breakdown Structure –

vehicle for budgeting and collecting costs. The software project manager must have insight into project costs and how they are expended. If the WBS is structured improperly, it can drive the evolving design in the wrong direction.

Business Case –

provides all the information necessary to determine whether the project is worth investing in. It details the expected revenue, expected cost, technical and management plans.

Release Specifications

Typical release specification outline :

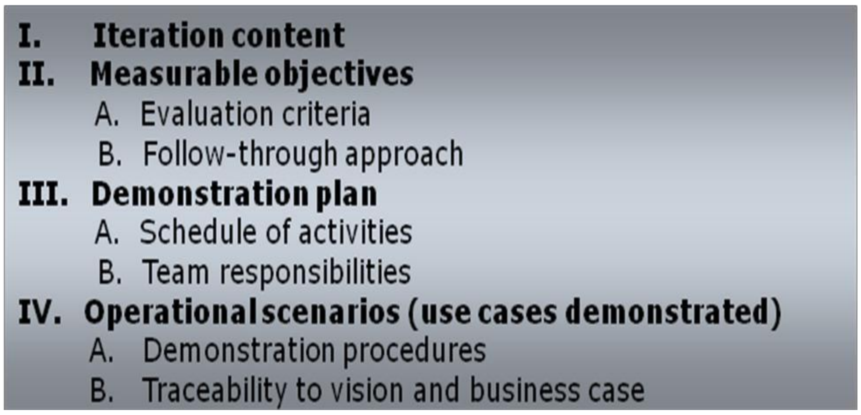

I. **Iteration content**
II. **Measurable objectives**
   A. Evaluation criteria
   B. Follow-through approach
III. **Demonstration plan**
   A. Schedule of activities
   B. Team responsibilities
IV. **Operational scenarios (use cases demonstrated)**
   A. Demonstration procedures
   B. Traceability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

Figure 2-7: Specification Outline

Two important forms of requirements:

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

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

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

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

 Environment – A robust development environment must support automation of the development process.

It should include:

- requirements management
- visual modeling
- document automation
- automated regression testing
-

Engineering Artifacts:

In general review, there are three *engineering* artifacts:

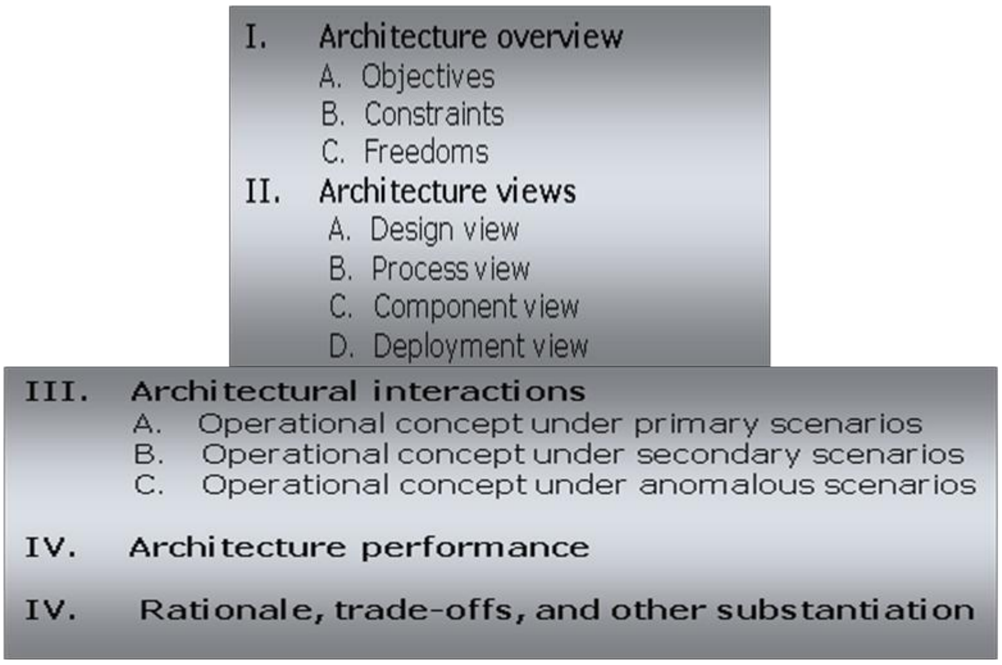
Vision document – supports the contract between the funding authority and the development organization.

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

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

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

Typical architecture description outline :



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

Figure 2-8: Architecture Overview

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

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

2.8 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:

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

Table 2-1: Life Cycle Aspects

Life - Cycle Aspect	**Engineering Stage Emphasis**	**Production Stage Emphasis**
Risk reduction	Schedule, technical feasibility	Cost
Products	Architecture baseline	Product release baselines

Activities	Analysis, design, planning	Implementation, testing
Assessment	Demonstration, inspection, analysis	Testing
Economics	Resolving diseconomies of scale	Exploiting economics of scale
Management	Planning	Operations

The engineering stage is decomposed into two distinct phases, inception and elaboration, and the production stage into construction and transition. These four phases of the life-cycle process are loosely mapped to the conceptual framework of the spiral model.
The size of the spiral model corresponds to the inertia of the project with respect to the breadth and depth of the artifacts that have been developed.

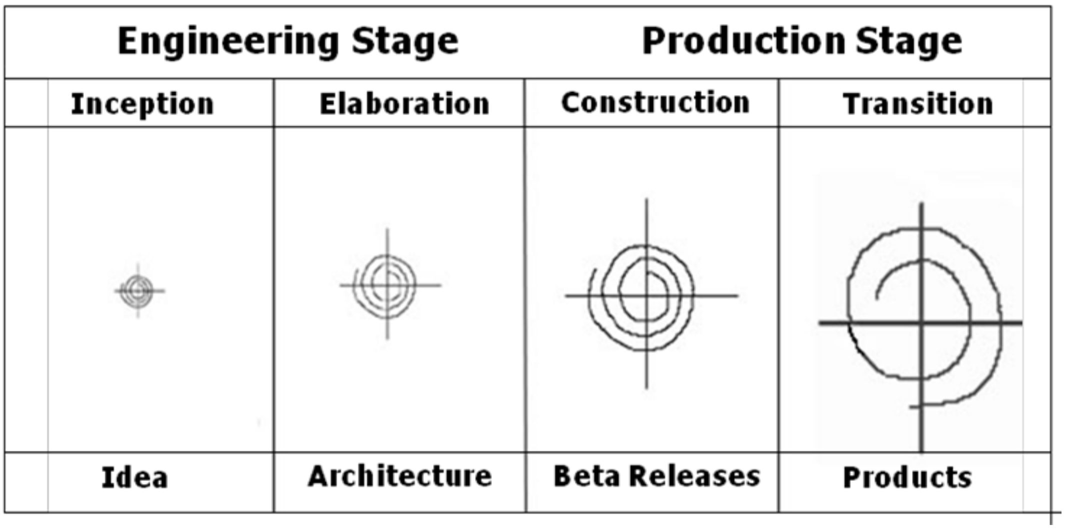


Figure 2-9: Life Cycle Stages

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.

2.8 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 scenanos
Estimating the cost and schedule for the entire project (including detailed estimates for the elaboration phase)
Estimating potential risks (sources of unpredictability)

Essential Actmties

Formulating the scope of the project. The information repository should be sufficient to define the problem space and derive the acceptance criteria for the end product.
Synthesizing the architecture. 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, and cost/schedule/profitability trade-offs are evaluated.

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? (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

2.9 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?

2.10 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?

2.11 Transition Phase

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

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

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

Primary Objectives

- 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?

2.12 The Artifact Sets

To make the development of a complete software system manageable, distinct collections of information are organized into artifact sets. A*rtifact* 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).

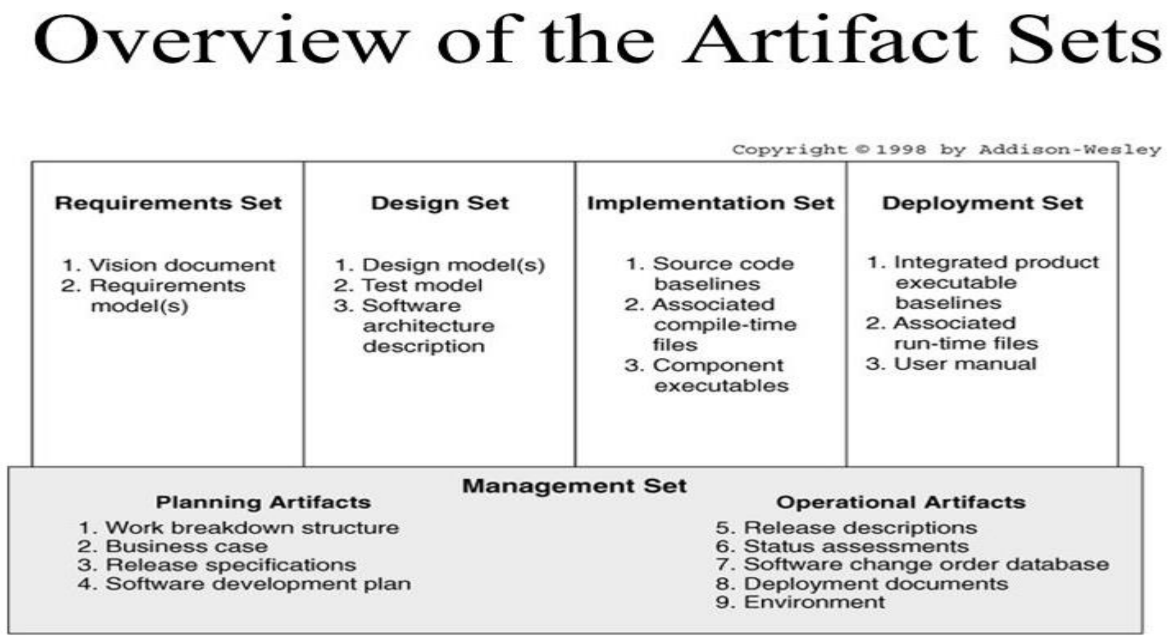


Figure 2-10: Overview of Artifacts Sets

2.13 The Management Artifacts

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 docu- ments (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

2.14 The Engineering Artifacts

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

Requirements Set

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

Analysis of consistency with the release specifications of the management set
Analysis of consistency between the vision and the requirements models

Mapping against the design, implementation, and deployment sets to evaluate the consistency and completeness and the semantic balance between information in the different sets

Analysis of changes between the current version of requirements artifacts and previous versions
(scrap, rework, and defect elimination trends)
Subjective review of other dimensions of quality Design Set
UML notation is used to engineer the design models for the solution. The design set contains varying levels of abstraction that represent the components of the solution space (their identities, attributes, static relationships, dynamic interactions). The design set is evaluated, assessed, and measured through a combination of the following Analysis of the internal consistency and quality of the design model
Analysis of consistency with the requirements models Translation into implementation and deployment sets and notations (for example, traceability, source code generation, compilation, linking) to evaluate the consistency and completeness and the semantic balance between information in the sets
Analysis of changes between the current version of the design model and previous versions (scrap, rework, and defect elimination trends)Subjective review of other dimensions of quality

Implementation set

The implementation set includes source code (programming language notations) that represents the tangible implementations of components (their form, interface, and dependency relationships)
Implementation sets are human-readable formats that are evaluated, assessed, and measured through a
combination of the following:

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

Deployment Set

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

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

- Testing against the usage scenarios and quality attributes defined in the requirements set to evaluate the consistency and completeness and the~ semantic balance between information in the two sets
- Testing the partitioning, replication, and allocation strategies in mapping components of the implementation set to physical resources of the deployment system (platform type, number, network topology)
- Testing against the defined usage scenarios in the user manual such as installation, user-oriented dynamic reconfiguration, mainstream usage, and anomaly management
- Analysis of changes between the current version of the deployment set and previous versions
- (defect elimination trends, performance changes)
- Subjective review of other dimensions of quality
- Each artifact set is the predominant development focus of one phase of the life cycle; the other sets take on check and balance roles. As illustrated in Figure 6-2, each phase has a predominant focus: Requirements are the focus of the inception phase; design, the elaboration phase; implementation, the construction phase; and deployment, the transition phase. The management artifacts also evolve, but at a fairly constant level across the life cycle.

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

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

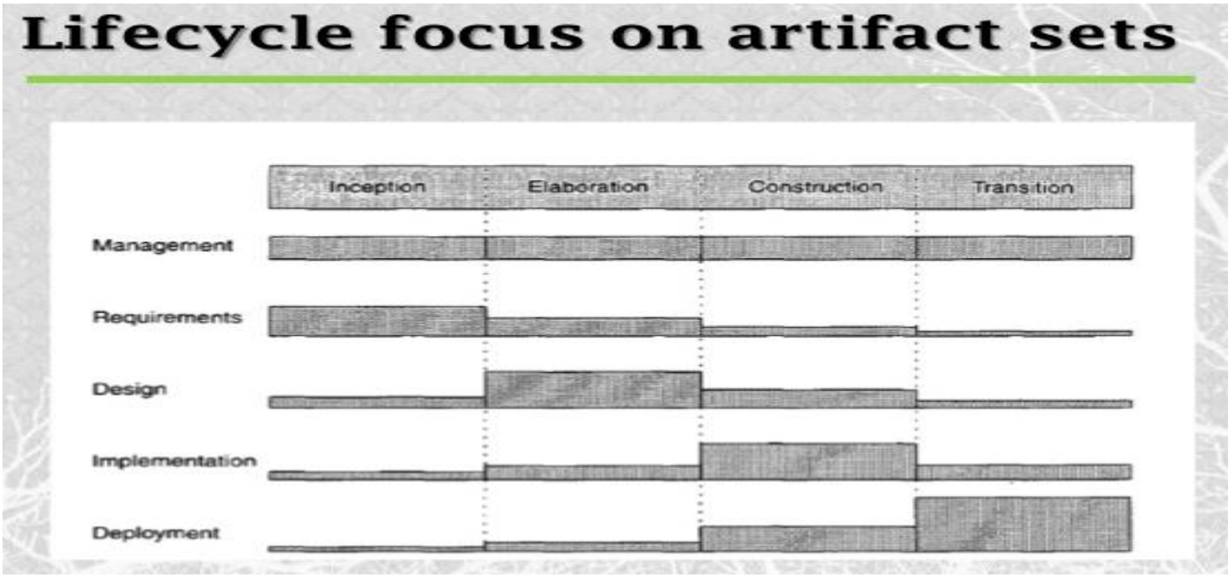


Figure 2-11: Lifecycle focus on artifacts sets

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

2.15 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.

2.16 Model-Based Software Architectures

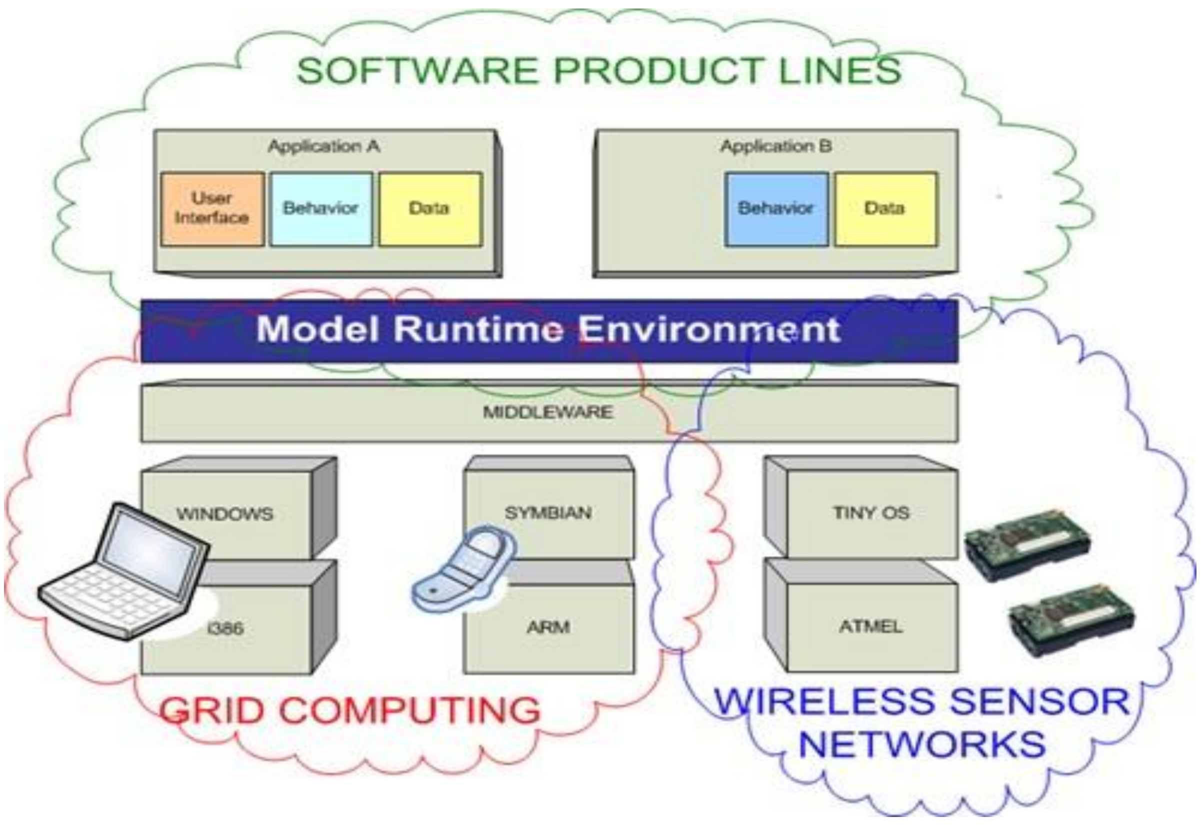


Figure 2-12: Model-Based Software Architectures

Models of software are used in an increasing number of projects to handle the complexity of application domains.

MODEL-BASED SOFTWARE ARCHITECTURES

- Models of software are used in an increasing number of projects to handle the complexity of application domains.
- By applying models for the specification of software application domain specific knowledge is separated from technological specific knowledge.
- While models can be applied in different phases of a software development process, research projects in the Model-based software architectures group are looking at the following application of models
- Generative approaches for tackling problems in the research area of software product lines

- Interpreting approaches to enable software architectures for Mobile Grid (and Cloud) Computing

- Architecture is the software system design.

- The ultimate goal of the engineering stage is to converge on a stable architecture baseline.

- An architecture baseline is not a paper document, it is a collection of information across all the engineering sets.

  - Architectures are described by extracting the essential information from the design models.
  - Software architecture is the central design problem of a complex software system.
  - However, like a architecture of a large building, the critical performance attributes and features of a complex software system cannot be described through stable laws of physics.

The Importance of s/w architecture and its close linkage with modern s/w development processes

- Achieving stable s/w architecture represents a significant project milestone at which the critical make/buy decisions should have been resolved.
- Architecture representations provide a basis for balancing the trade-offs between the problem space and the solution space.
- The architecture and process encapsulate many of the important communications among individuals, teams, organizations, and stakeholders.
- Poor architectures and immature processes are often given as reasons for project failures.

Software architecture encompasses the structure of software systems, their behavior, and the patterns that guide these elements, their collaboration, and their composition.
The context of s/w architecture structure, behavior, and patterns must include functionality, performance, resilience, comprehensibility, economic trade-offs, technology constraints and aesthetic concerns.
An architecture framework is defined in terms of views that are abstractions of the UML models in the design set.
An architecture view is an abstraction of the design model; it contains only the architecturally significant information.

Four Views

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.

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 components inventory and bill of materials of all primitive components

Deployment: Executable components sufficient to demonstrate the critical use cases and the risk associated with achieving the system qualities.

An architecture baseline is defined as a balanced subset of information across all sets, whereas an architecture description is completely encapsulated within the design set.
The architecture description will take a wide range of forms, from a simple, direct subset of UML diagrams to a complex set of models with a variety of distinct views that capture and compartmentalize the concerns of a sophisticated system.