ESPM UNIT I

Conventional Software Management: The Waterfall Model, Conventional software Management Performance.
Evolution of Software Economics: Software Economics, Pragmatic Software Cost Estimation.
Improving Software Economics: Reducing Software Product Size, Improving software Processes, Improving Team Effectiveness, Improving Automation, Achieving Required Quality, Peer Inspections.

INTRODUCTION

- Conventional software Management Practices ear sound in theory, but practice is still tied to archaic 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:

- Software development is still highly unpredictable. Only about 10% of software projects are delivered successfully within initial budget and schedule estimates.
- Management discipline is more of a discriminator in success or failure than are technology advances.
- The level of software scrap and rework is indicative of an immature process.

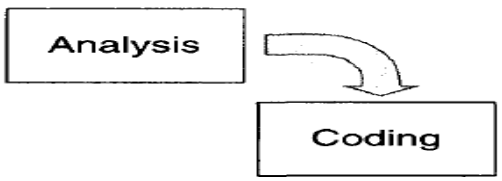
All three analyses reached the same general conclusion: The success rate for software projects is very low. The three analyses provide a good introduction to the magnitude of the software problem and the current norms for conventional software management performance.

1.1 THE WATERFALL MODEL (IN THEORY)

Most software engineering texts present the waterfall model as the source of the **"conventional"** software process. In 1970, Winston Royce presented a paper called "Managing the Development of Large Scale Software Systems" at IEEE WESCON, where he made three primary points:

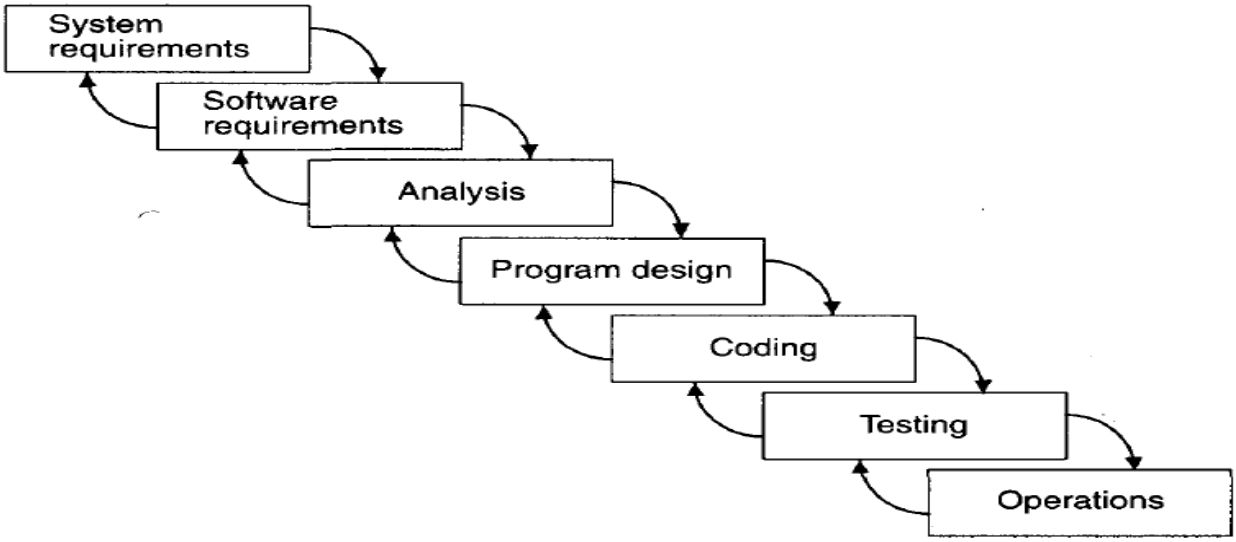
1. There are two essential steps common to the development of computer programs: analysis and coding.
2. In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other "overhead" steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps.
3. The basic framework described in the waterfall model is risky and invites failure. The testing phase that occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished form analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violate. Either the requirements must be modified or a substantial design change is warranted.

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



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

Waterfall Model Part 2 : The large-scale system approach



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

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

Five necessary improvements for waterfall model are (the *risks may be eliminated by making the following five improvements*):-

(a) Program design comes first: The first step is to insert a preliminary program design phase between the software requirements phase and the analysis phase. Hence, by this technique, the software failure will not occur due to the continuous change in storage, timing and data. The designer then urges the storage, timing and operational limitations. On the analyst in such a way, that he notices the results. Resources insufficiently and the design limitations are identified in the early stages before final designing coding and testing. The following steps are required:

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

(b) Document the design. The amount of documentation associated with the software programs is very large because of the following reasons:

(a) Each designer must communicate with interfacing designers, managers, and possibly customers.
(b) During early phases, the documentation is the design.
(c) The real monetary value of documentation is to support later modifications by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

(c) Do it twice. The Computer program must be developed twice and the second version, which takes into account all the critical design operations, must be finally delivered to the customer for operational development. The first version of the computer program involves a special board competence team, responsible for notifying the troubles in design, followed by their modeling and finally generating an error-free program.

(d) Plan, control, and monitor testing. The test phase is the biggest user of the project resources, such as manpower, computer time and management assessments. It has the greatest risk in terms of cost and schedules and develops at the most point in the schedule, when backup alternatives are least available. Thus, most of the problems need to be solved before the test phase, as it has to perform some other important operations.
(a) Hire a team of test specialists, who are not involved in the original design.
(b) Apply visual inspections to discover the obvious errors, such as skipping the wrong addresses, dropping of minus signs etc.
(c) Conduct a test for every logic path;
(d) Employ the final checkout on the target computer.

(e) **Involve the customer.** The customer must be involved in a formal way, so that, he has devoted himself at the initial stages before final delivery. The customer‟s perception, assessment and commitment can strengthen the development effort. Hence, an initial design step followed by a "preliminary software review", "critical software design reviews", during design and a "final software acceptance review" after testing is performed.

THE WATERFALL MODEL (IN PRACTICE)

- Some software projects still practice the conventional software management approach.
- It is useful to summarize the characteristics of the conventional process as it has typically been applied, which is not necessarily as it was intended. Projects destined for trouble frequently exhibit the following symptoms:
  (a) Protracted integration and late design breakage
  (b) Late risk resolution
  (c) Requirements – driven functional decomposition
  (d) Adversarial (conflict or opposition) stakeholder relationships
  (e) Focus on documents and review meetings.

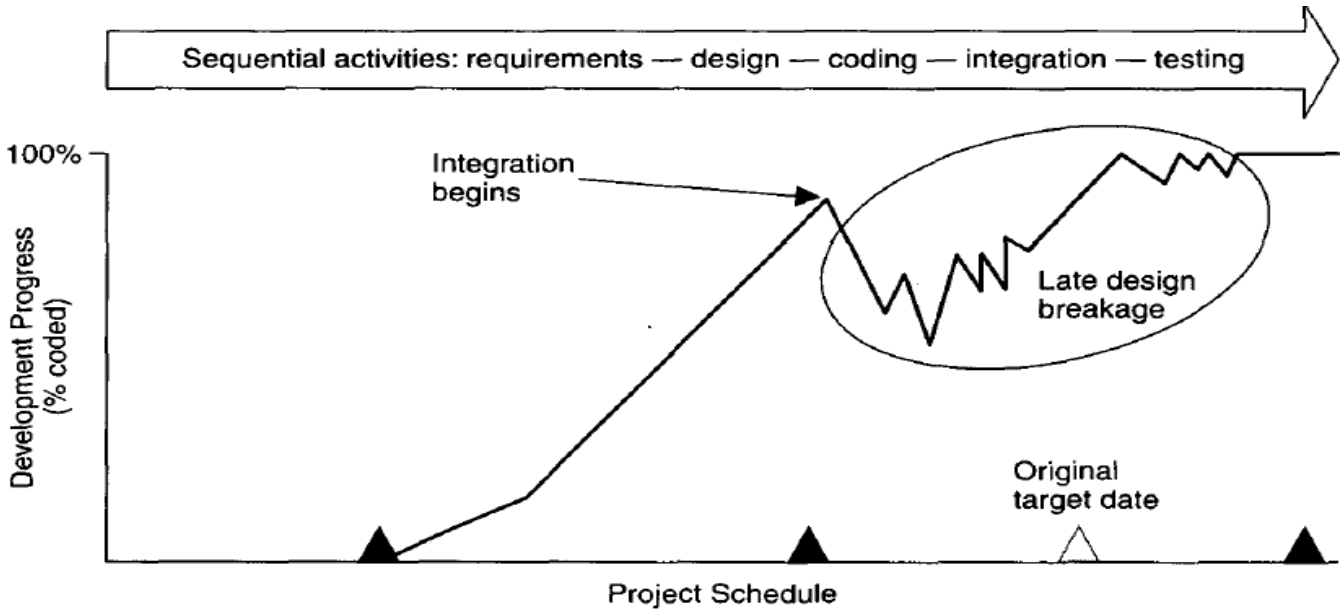
(a) Protracted Integration and Late Design Breakage

The percentage of progress achieved in the development process against time is shown in Figure.
The following sequence was common:

- Early success via paper designs and thorough (often *too* thorough) briefings
- Commitment to code late in the life cycle
- Integration nightmares (unpleasant experience) due to unforeseen implementation issues and inter
- Heavy budget and schedule pressure to get the system working
- Late shoe-homing of no optimal fixes, with no time for redesign
- A very fragile, unmentionable product delivered late

Hence, when a waterfall model is used in the process, late integration and performance degradation occurs.

Format	Ad hoc text	Flowcharts	Source code	Configuration baselines
Activity	Requirements analysis	Program design	Coding and unit testing	Protracted integration and testing
Product	Documents	Documents	Coded units	Fragile baselines



Progress profile of a conventional software project

Expenditures by activity for a conventional software project

ACTIVITY	COST
Management	5%
Requirements	5%
Design	10%
Code and unit testing	30%
Integration and test	40%
Deployment	5%
Environment	5%
Total	100%

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

(b) Late risk resolution

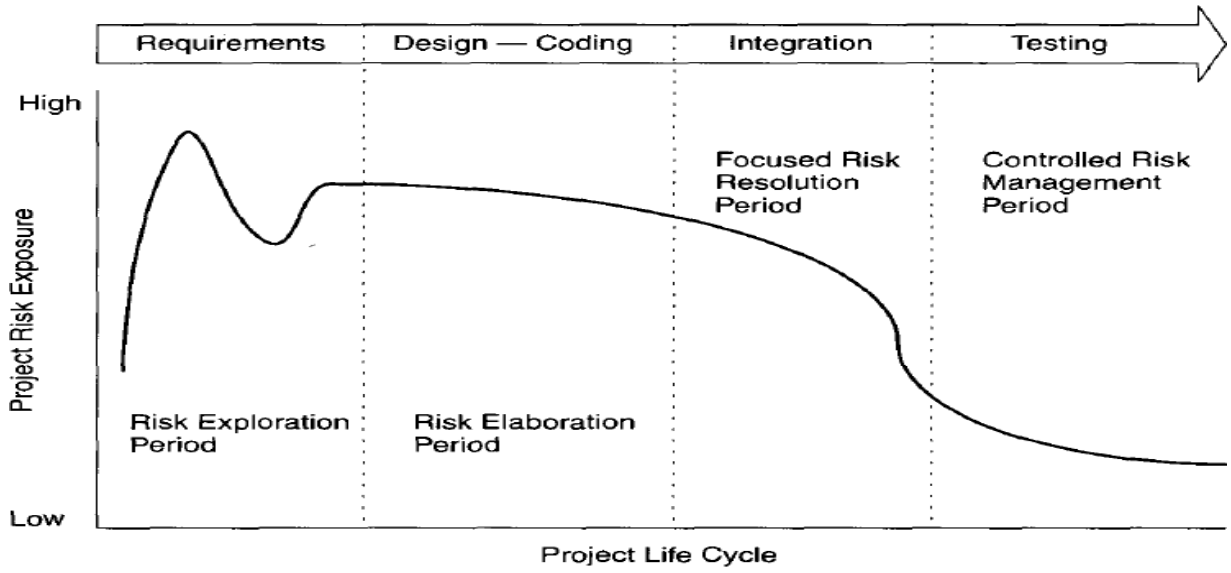
A serious issue associated with the waterfall lifecycle was the lack of early risk resolution.

A ***risk*** is defined as the probability of missing a cost, schedule, feature, or quality goal.

The Figure illustrates a typical risk profile for conventional waterfall model projects. It includes four distinct periods of risk exposure.

- Early in the life cycle, as the requirements· were being specified, the actual risk exposure was highly unpredictable.
- After a design concept is available to balance the understanding the requirements, the risk exposure is stabilized.

- When integration began, during this period real design issues were resolved and engineering tradeoffs were made.



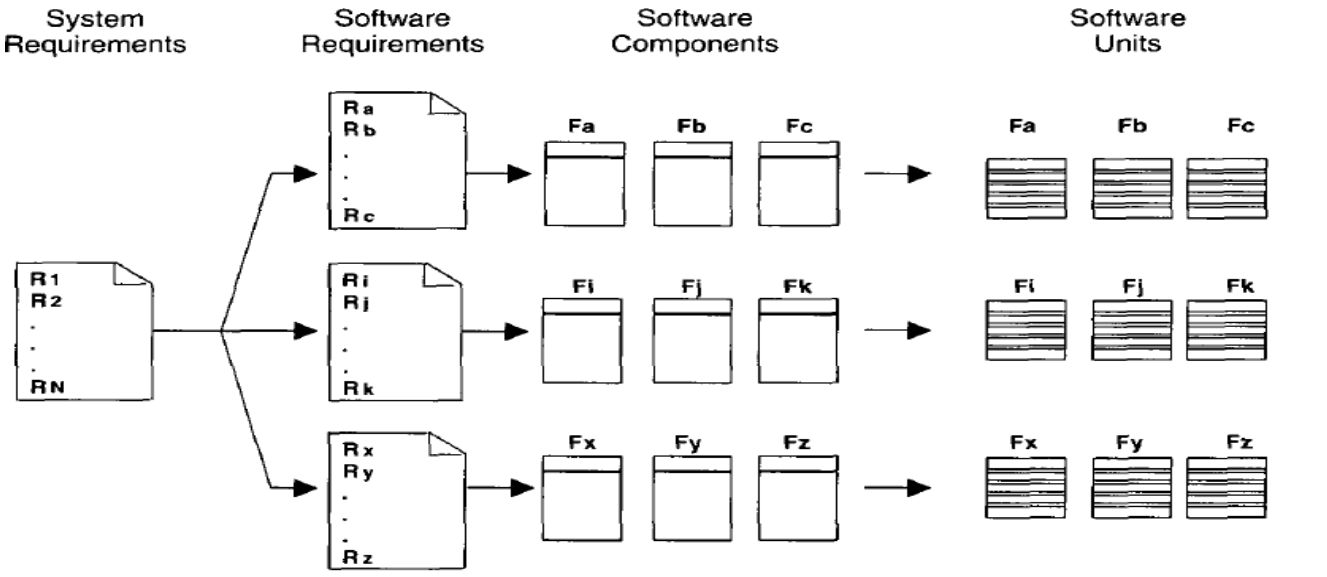
Risk profile of a conventional software project across its life cycle

(c) Requirements – driven functional decomposition

The software development process is requirements driven i.e., initially gives precise definition for the requirements and then provides implementation for them.

The requirements are specified completely and clearly before any other activities in the software development process. It immaturely treats all the requirements, equally. Requirements specification is an important and difficult job in the development process

The basic assumption of the waterfall process is that, requirements are specified in a functional manner, according to when the software can be divided into functions followed by the requirements allocation to the resulting components.



Suboptimal software component organization resulting from a requirements-driven approach

(d) Adversarial (conflict or opposition) stakeholder relationships

The conventional process tended to result in adversarial stakeholder relationships, in large part because of the difficulties of requirement specification and the exchange of information solely through paper documents that captured engineering information in ad hoc formats.

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

(a) The contractor prepared a draft contract-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.

(b) The customer was expected to provide comments (typically within 15 to 30 days).

(c) The contractor incorporated these comments and submitted (typically within 15 to 30 days) a final version for approval.

This one-shot review process encouraged high levels of sensitivity on the part of customers and contractors.

(e) Focus on documents and review meetings.

Emphasis on documents generation while describing a software product causes insufficient focus on producing tangible product increments.

Implementation of the major milestones can be done through documents specification. Contractors, rather than reducing the risks to improve the product quality, produces large amount of paper for creating the documents and only the simple things are reviewed.

Hence, most design reviews have low engineering value and high costs in terms of schedule and effort.

Results of conventional software project design reviews

APPARENT RESULTS	REAL RESULTS
Big briefing to a diverse audience	Only a small percentage of the audience understands the software. Briefings and documents expose few of the important assets and risks of complex software systems.
A design that appears to be compliant	There is no tangible evidence of compliance. Compliance with ambiguous requirements is of little value.
Coverage of requirements (typically hundreds)	Few (tens) are design drivers. Dealing with all requirements dilutes the focus on the critical drivers.
A design considered "innocent until proven guilty"	The design is always guilty. Design flaws are exposed later in the life cycle.

1.2. CONVENTIONAL SOFTWARE MANAGEMENT PERFORMANCE

Barry Boehm"s "***Industrial software Metrics top 10 List***" is a good, objective characterization of the state of software development.

1) Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.

2) You can compress software development schedules 25% of nominal (small), but no more.

3) For every $1 you spend on development, you will spend $2 on maintenance.

4) Software development and maintenance costs are primarily a function of the number of source lines of code.
5) Variations among people account for the biggest difference in software productivity.
6) The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.
7) Only about 15% of software development effort is devoted to programming.
8) Software systems and products typically cost 3 times as much per SLOC as individual software programs. Software-system products cost 9 times as much.
9) Walkthroughs catch 60% of the errors.
10) **80/20 Principle:** 80% of the contribution comes from 20% of the contributors.
- 80% of the engineering is consumed by 20% of the requirements.
- 80% of the software cost is consumed by 20% of the components.
- 80% of the errors are caused by 20% of the components.
- 80% of the software scrap and rework is caused by 20% of the errors.
- 80% of the resources are consumed by 20% of the components.
- 80% of the engineering is accomplished by 20% of the tools.
- 80% of the progress is made by 20% of the people.

SOFTWARE ECONOMICS

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

- 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.
- 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).
- The capabilities of software engineering **personnel**, and particularly their experience with the computer science issues and the applications domain issues of the project.
- The **environment**, which is made up of the tools and techniques available to support efficient software development and to automate the process
- The required **quality** of the product, including its features, performance, reliability and adaptability.

The relationships among these parameters and the estimated costs can be written as follows:

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

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.

The three generations or 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.

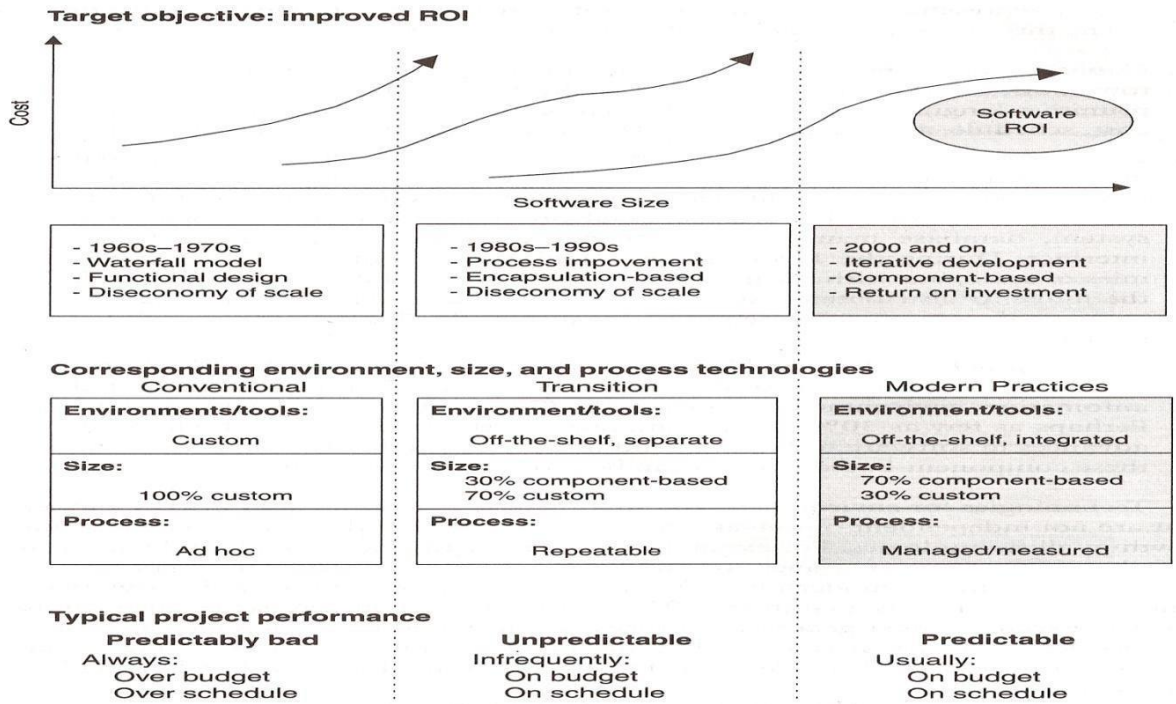


FIGURE 2-1. *Three generations of software economics leading to the target objective*

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.

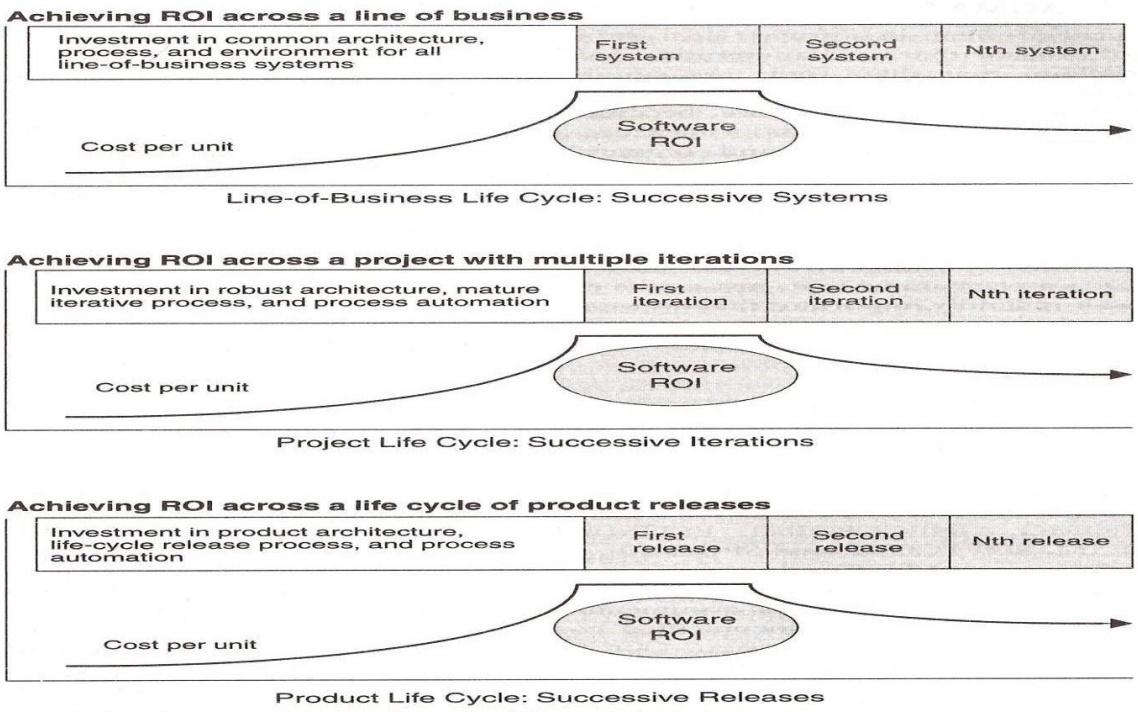


FIGURE 2-2. *Return on investment in different domains*

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), CO COMO 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 a "within 20% of actual, 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 may be 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.

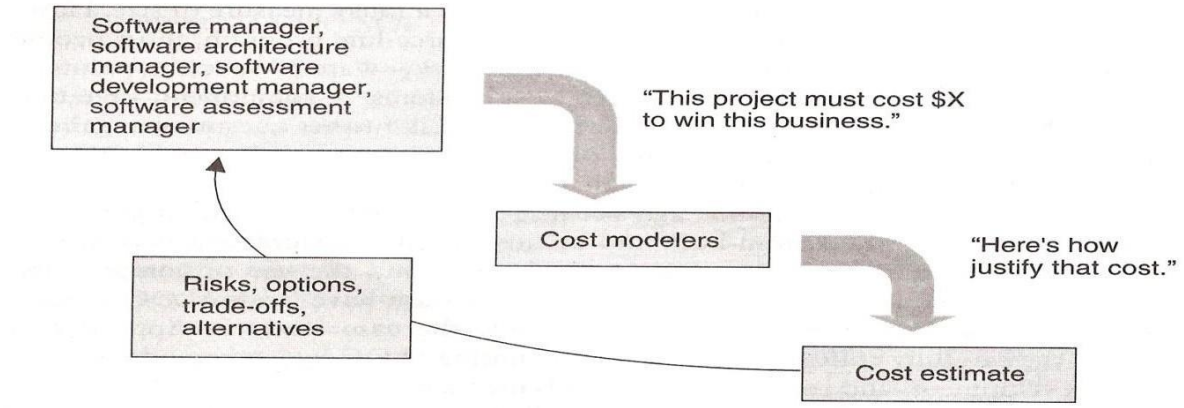


FIGURE 2-3. *The predominant cost estimation process*

A good software cost estimate has the following attributes:

- It is conceived and supported by the project manager, architecture team, development team and test 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 similarprocesses, similar technologies, similar environments, similar quality requirements and similar people.
- It is define din enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

INTRODUCTION

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

Table: Important trends in improving software economics

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

REDUCING SOFTWARE PRODUCT SIZE

- 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 here as the general term for reducing the "source" language size necessary to achieve a software solution.
- Reuse, object oriented technology, automatic code production, and higher order programming languages are all focused on achieving a given system with fewer lines of human-specified source.

- This size reduction is the primary motivation behind improvements in higher order languages (such as C++, Ada 95, Java, Visual Basic, and fourth-generation languages), 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).
- In general, when size-reducing technologies are used, they reduce the number of human-generated source lines.

LANGUAGES

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

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

a) Visual basic is very powerful and expressive in building simple interactive applications but it would not be used for real time, embedded.
b) Ada 95 might be the best language for a catastrophic, cost of failure system that controls nuclear power plant but it would not be used for highly parallel, scientific, number crunching program running on a super computer.
c) Ada 83 is used by Department of Defense (DOD) to increase it would provide it expressiveness.
d) C++ incorporated several advances with in Ada as well as advanced support for object oriented programming.
e) C compatability made easy for C programmer to transition to C++
f) The evolution of Java has eliminated many problems in C++, while conserving object oriented features and adding further support for portability and support.
UPFs (Universal Function Points) are useful estimators for language-independent in the early life cycle phases.

1,000,000 lines of assembly language
400,000 lines of C
220,000 lines of Ada 83
175,000 lines of Ada 95 or C++

The values indicate the relative expressiveness provided by various languages.

OBJECT-ORIENTED METHODS AND VISUAL MODELING

- There has been at widespread movement in the 1990s toward object-oriented technology. The advantages of object-oriented methods include improvement in software productivity and software quality. The fundamental impact of object-oriented technology is in reducing the overall size of what needs to be developed.

Booch describes the following <u>three</u> reasons for the success of the projects that are using Object Oriented concepts:

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 <u>five</u> 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.
- Software design methods have always dealt implicitly with reuse in order to minimize development costs while achieving all the other required attributes of performance, feature set, and quality.
- Most truly reusable components of value are transitioned to commercial products supported by organizations with the following characteristics:
  1. They have an economic motivation for continued support.
  2. They take ownership of improving product quality, adding new features, and transitioning to new technologies.
  3. They have a sufficiently broad customer base to be profitable.
- Reuse is an important discipline that has an impact on the efficiency of all workflows and the quality of most artifacts. The cost of developing a reusable component is not trivial. The following Figure examines the economic tradeoffs. The steep initial curve illustrates the economic obstacle to developing reusable components.

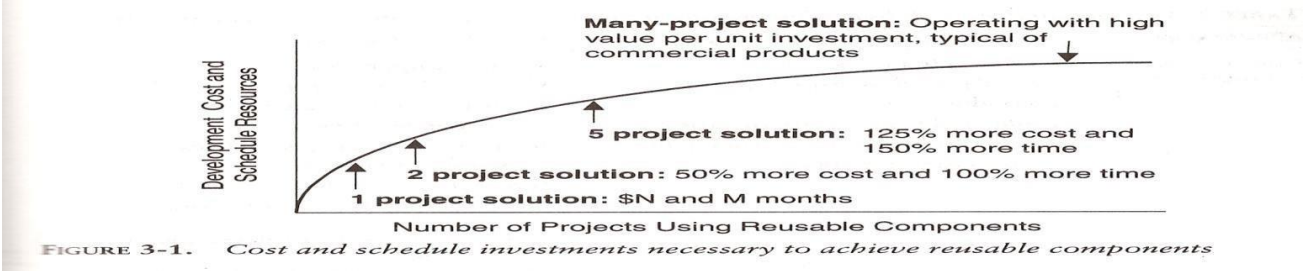


FIGURE 3-1. *Cost and schedule investments necessary to achieve reusable components*

COMMERCIAL COMPONENTS

- A common approach being pursued today in may 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 straight forward in practice.
- The following Table identifies some of the advantages and disadvantages of using commercial components.

APPROACH	ADVANTAGES	DISADVANTAGES
Commercial components	1. Predictable license costs 2. Broadly used, mature technology Available now 3. Dedicated support organization 4. Hardware/ Software independence 5. Rich in functionality	1. Frequent upgrades 2. Up-front license fees 3. Recurring maintenance fees 4. Dependency on vendor 5. Run-time efficiency sacrifices 6. Functionality constraints. 7. Integration not always trivial 8. No control over upgrades and maintenance 9. Unnecessary features that consume extra resources 10. Often Inadequate reliability and Stability. 11. Multiple-vendor incompatibilities
Custom development	1. Complete change freedom 2. Smaller, often simpler implementations 3. Often better performance 4. Control of development and enhancement	1. Expensive, unpredictable development 2. Unpredictable availability date 3. Undefined maintenance model 4. Often immature and fragile 5. Single-platform dependency 6. Drain on expert resources

IMPROVING SOFTWARE PROCESSES

- Process is an overloaded term. For software-oriented organizations, there are many processes and sub processes. Three distinct process perspectives are:
  1. **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.
  2. **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.
  3. **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.

Table: three levels of process and their attributes

ATTRIBUTES	METAPROCESS	MACROPROCESS	MICROPROCESS
Subject	Line of Business	Project	Iteration
Objectives	Line-of-business profitability Competitiveness	Project Profitability Risk management Project Budget, Schedule, quality	Resource Management Risk resolution Milestone budge, schedule, quality
Audience	Acquisition authorities, customers, organizational management	Software project managers Software engineers	Subproject Managers Software Engineers
Metrics	Project predictability Revenue, market share	On budget, on schedule Major milestone success Project scrap and rework	On budget, on schedule major milestone progress/ 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:
  1. A well-managed project can succeed with a nominal Engineering team.
  2. 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.
  3. A poorly architected system will flounder even with an expert team ofbuilders.
- **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.

3. **Decision-Making skill**: The jillion books written about management have failed to provide a clear definition of this attribute.
4. **Team- building skill**: Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.
5. **Selling skill**: Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives.

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 far evolving the software engineering artifacts.
- 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 andschedules, and overall productivity using a smaller team. Integrated toolsets play an increasingly important role in incremental/iterative development by allowing the designers to traverse quickly among development artifacts and keep them up-to-date.
- ***Round-trip Engineering*** is a term used to describe the key capability of environments thatsupport iterative development.
- ***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 codeinto executable code.
- ***Reverse engineering*** is the generation or modification of a more abstract representation from an existing artifact.
- 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:
  1. Requirements analysis and evolution activities consume 40% of life-cycle costs.
  2. Software design activities have an impact on more than 50% of the resources.
  3. Coding and unit testing activities consume about 50% of software development effort and schedule.
4. Test activities can consume as much as 50% of a project's resources.
5. Configuration control and change management are critical activities that can consume as much as 25% of resources on a large-scale project.
6. Documentation activities can consume more than 30% of project engineering resources.
7. 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. Key practices that improve overall software quality include the following:

1. ***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.
2. ***Using metrics and indicators*** to measure the progress and quality of architecture as it evolves from a high-level prototype into a fully compliant product.
3. ***Providing integrated life-cycle environments*** that support early and continuous configuration control, change management, rigorous design methods, document automation, and regression test automation.
4. ***Using visual modeling and higher level languages*** that support architectural control, abstraction, reliable programming, reuse, and self-documentation
5. ***Early and continuous insight*** into performance issues through demonstration-based evaluations.

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:

1. **Project Inception:** The proposed design was asserted to be low risk with adequate performance origin.
2. **Initial design review:** Optimistic assessments of adequate design margin were based mostly on paper analysis or ought simulation of the critical threads. In most cases, the actual application algorithms and database sizes were fairly well understood.
3. **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.
4. **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 poorlyunderstood early design trade-offs.

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:
  1. Transitioning Engineering information from one artifact set to another, thereby assessing the consistency, feasibility, understandability, and technology constraints inherent in theengineering artifacts.
  2. Major milestone demonstrations that force the artifacts to be assessed against tangible criteria in the context of relevant use cases
  3. Environment tools (compilers, debuggers, analyzers, automated test suites) that ensure representation rigor, consistency, completeness, and change control
  4. Life-cycle testing for detailed insight into critical trade-offs, acceptance criteria andrequirements compliance.
  5. 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.