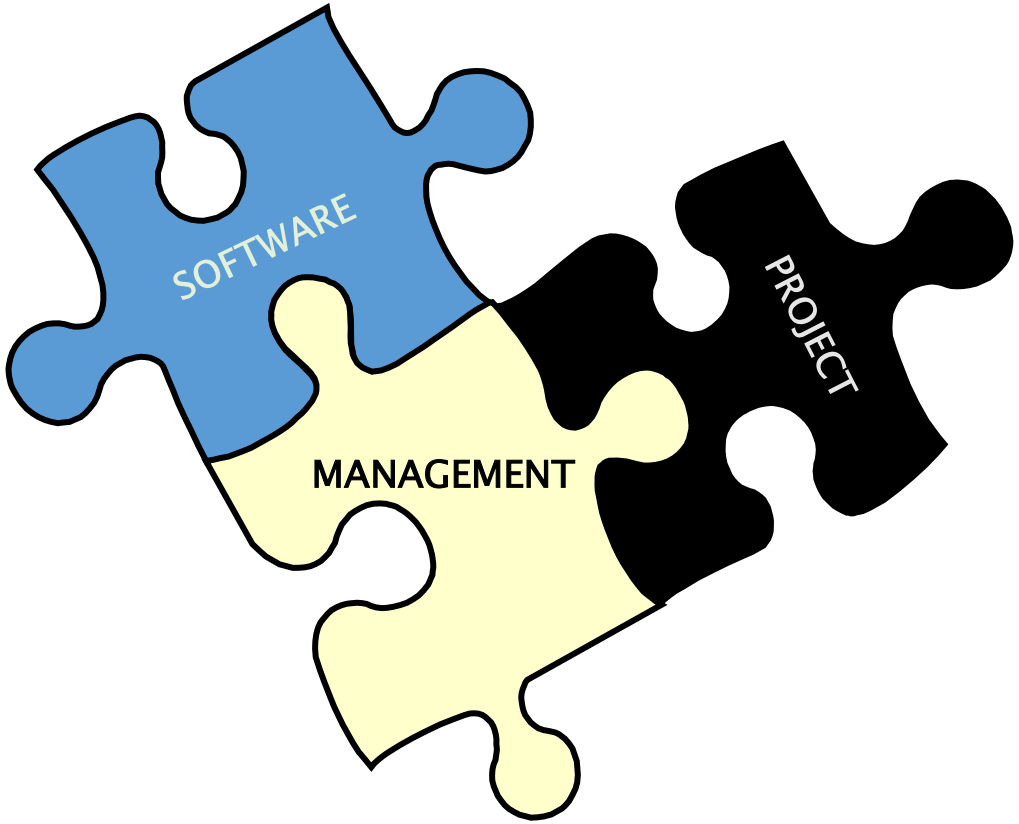
Software Project Management



Course Title: Software Project Management

- **Evaluation:** Sessional + Final
- **Marks**: 50+50
- **Nature of Course:** Theory

Course Objectives:

- To provide exposure to the students in the area of SPM as a new management framework uniquely suited to the complexities of modern software development process.
- To discuss the software engineering approaches to modern software development process (Unified Process).

Laboratory Works

Tools Required:

- Microsoft Project (2007 and/or higher versions)
- Microsoft Visio (2007 and/or higher versions)

Pre-requisite Courses

- System Analysis and Design
- Software Engineering
- Principle of Management
- Project Management

Course Contents: (Syllabus)

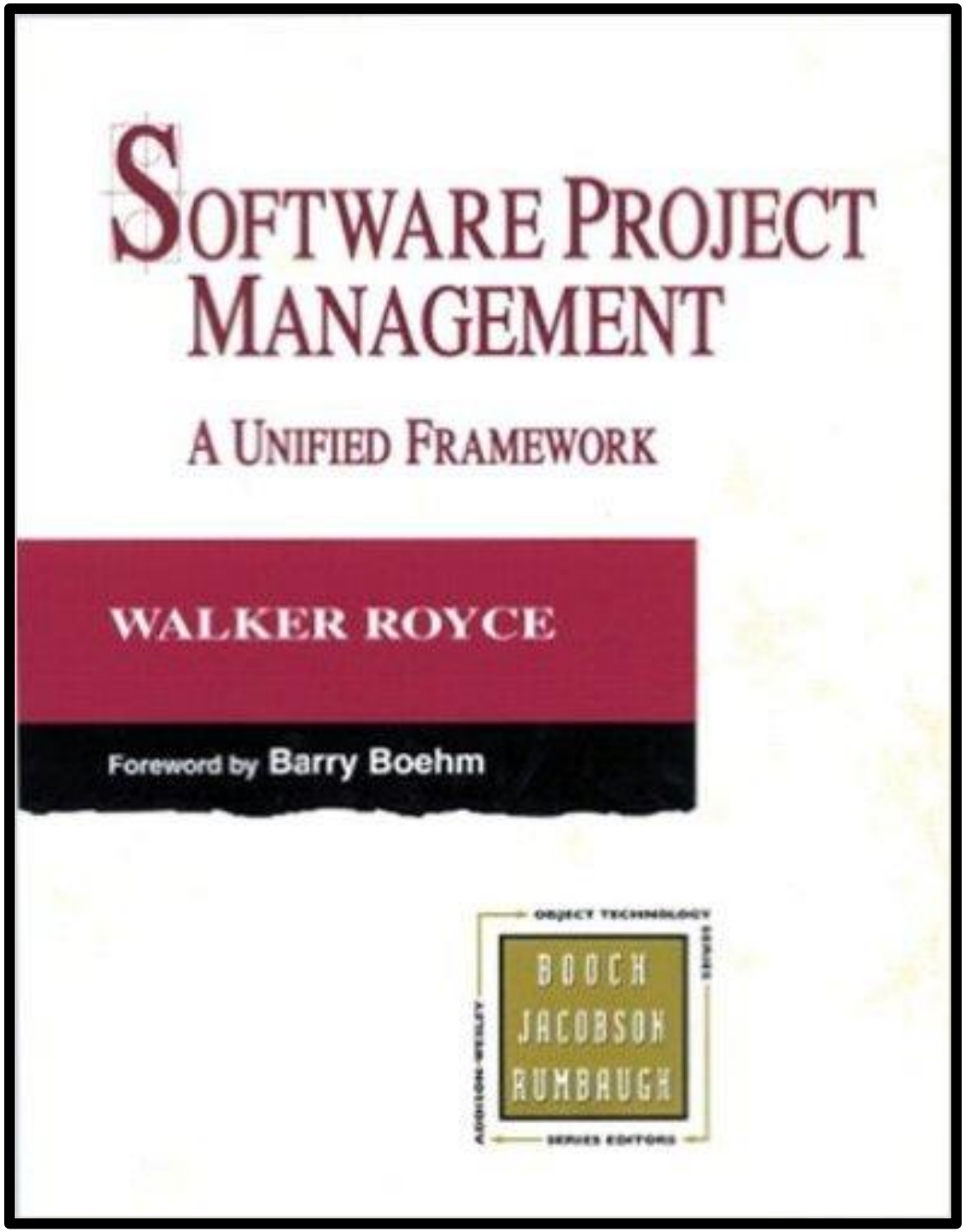
Unit 1. Software Management Practice and Software Economics – 12 Hrs.

Unit 2. Software Process, Primitives and Process Management Framework - 14 Hrs.

Unit 3. Techniques of Planning, Controlling and Automating Software Process - 15 Hrs.

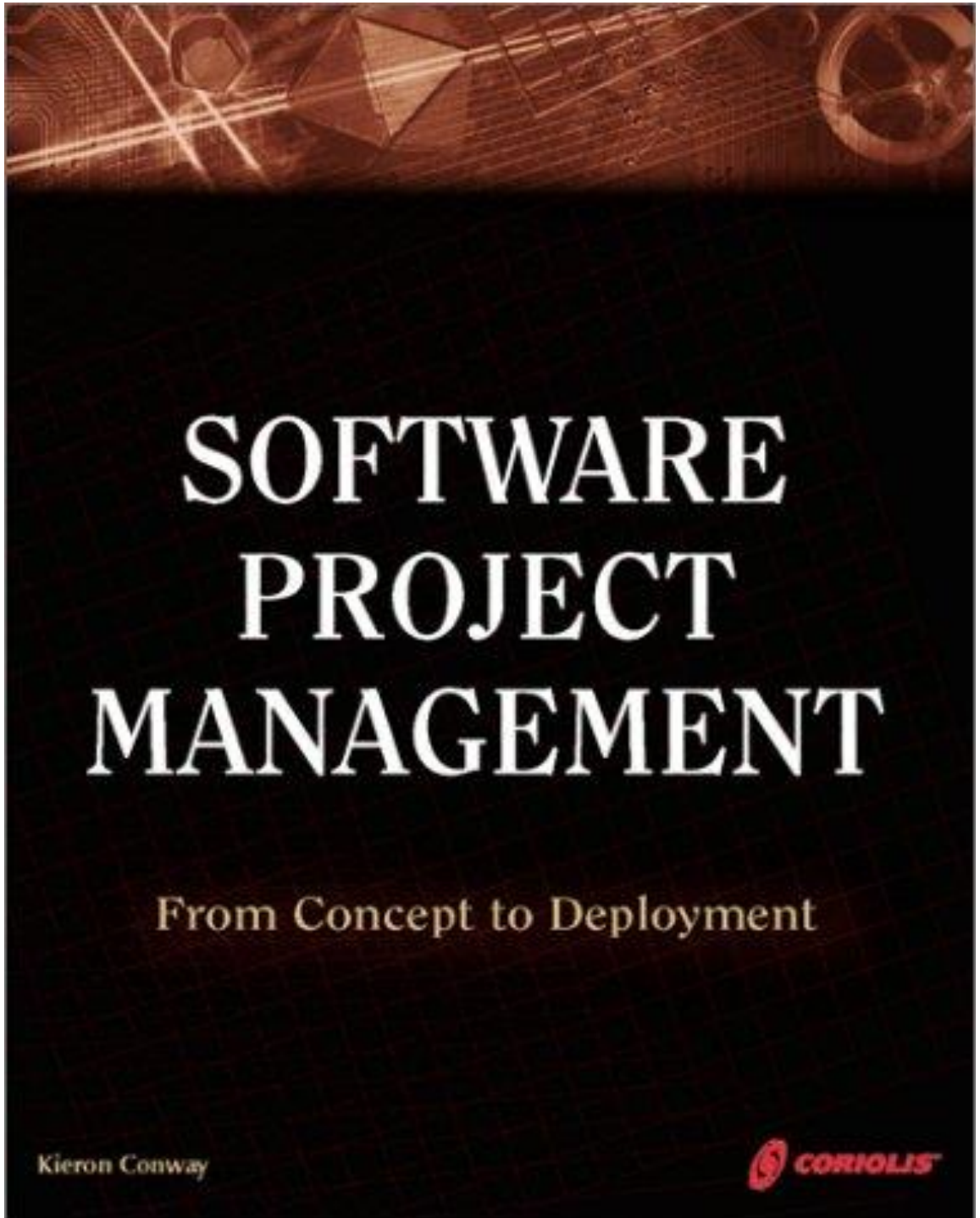
Unit 4. Modern Approach to Software Project and Economics - 4 Hrs.

Text Book



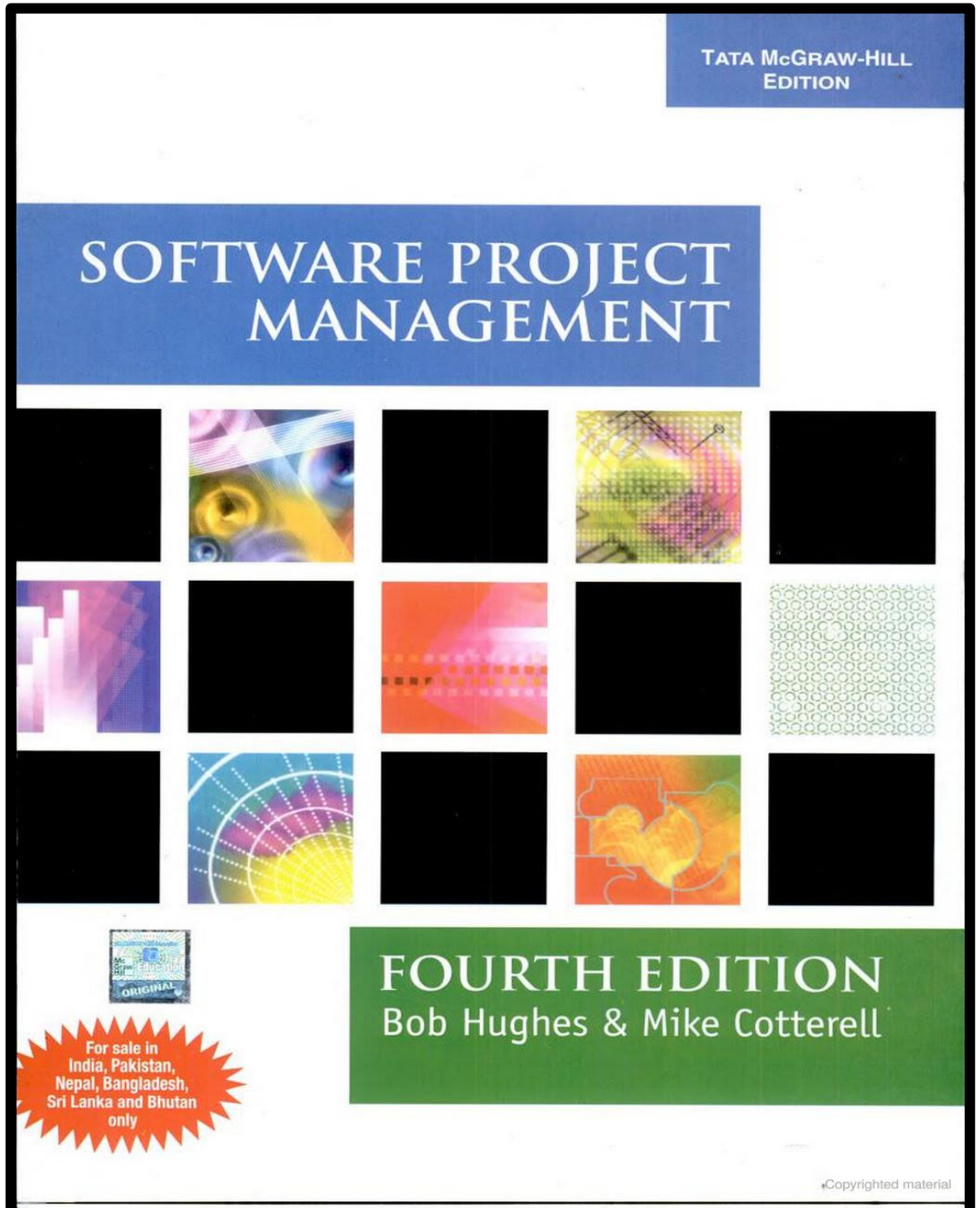
Text Book

Reference Books



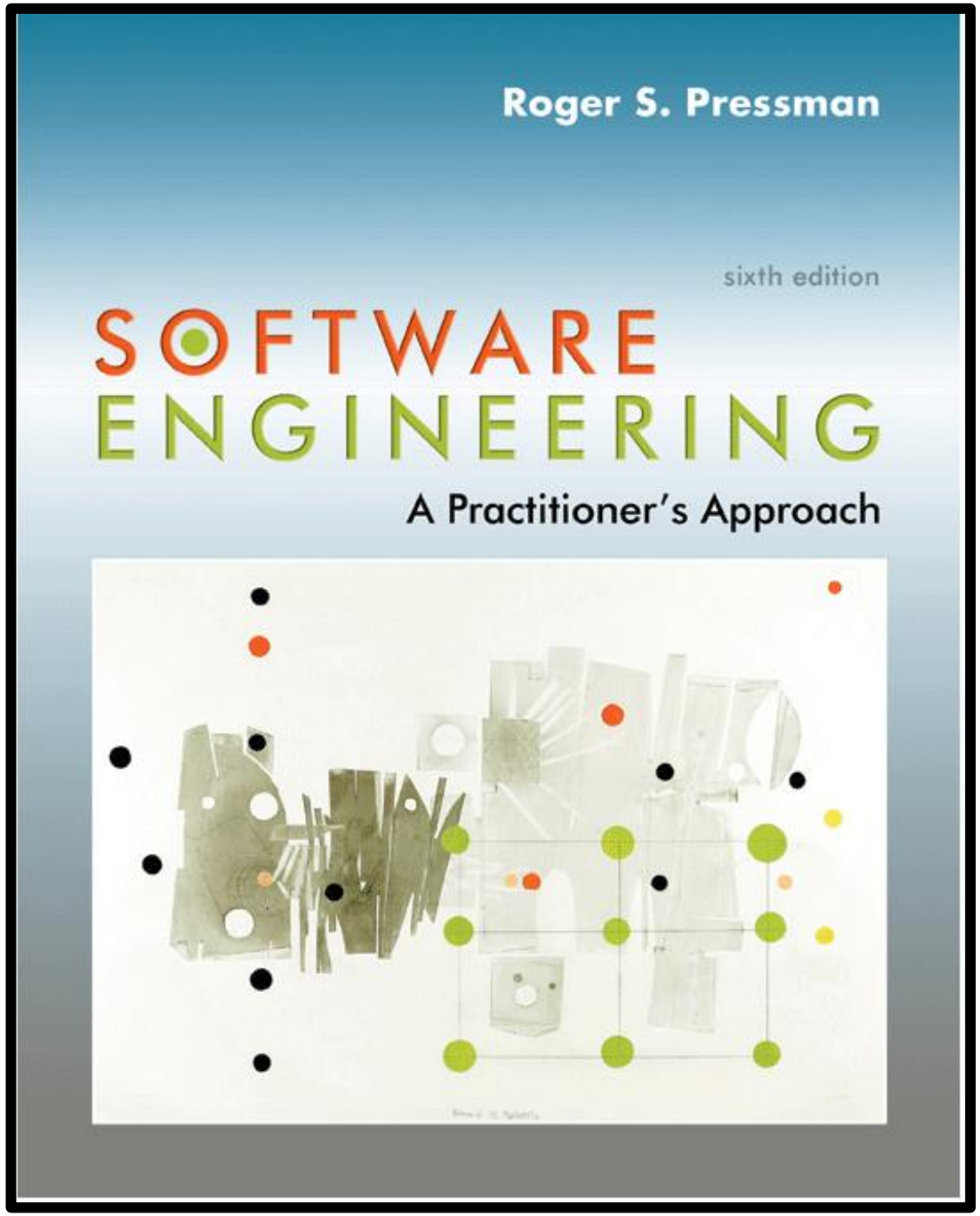
Reference Books

Reference Books



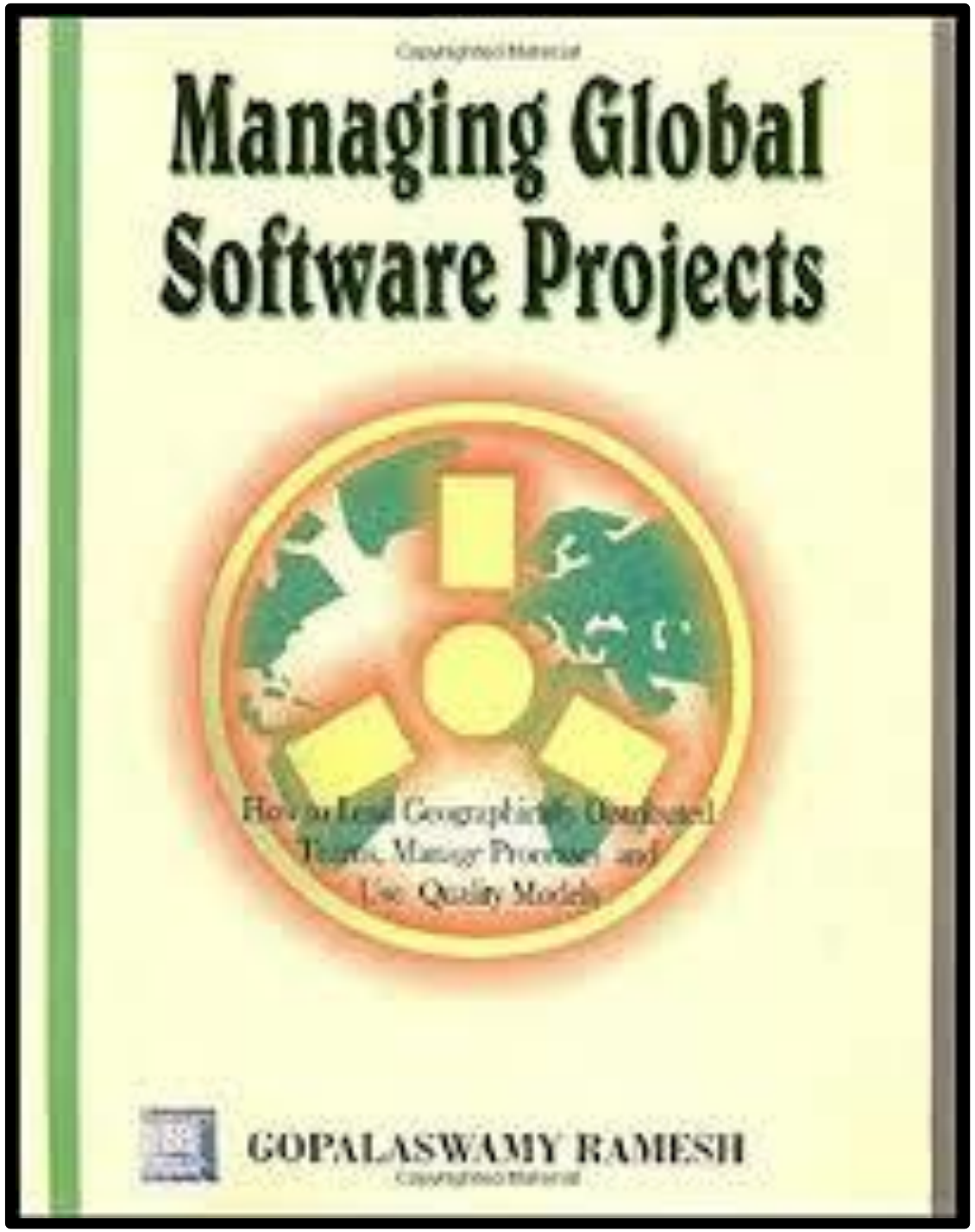
Reference Books

Reference Books



Reference Books

Reference Books



Reference Books

Reference Books

Reference Books

Unit 1. Software Management Practice and Software Economics

Prepared By:

BIJAY MISHRA

(बिजय मिश्र)

biizay@gmail.com

@jijibisha

UNIT 1: Software Management Practice and Software Economics - 12 Hrs.

1.1 Conventional software management theory and practice

1.2 Software economics and cost estimations

1.3 Improving software economics

1.4 Software process

1.5 Team effectiveness and software environment and quality target

1.6 Principles of Conventional software engineering

1.7 Principles of modern software management

1.8 Iterative process

Definition of a Software Project Management

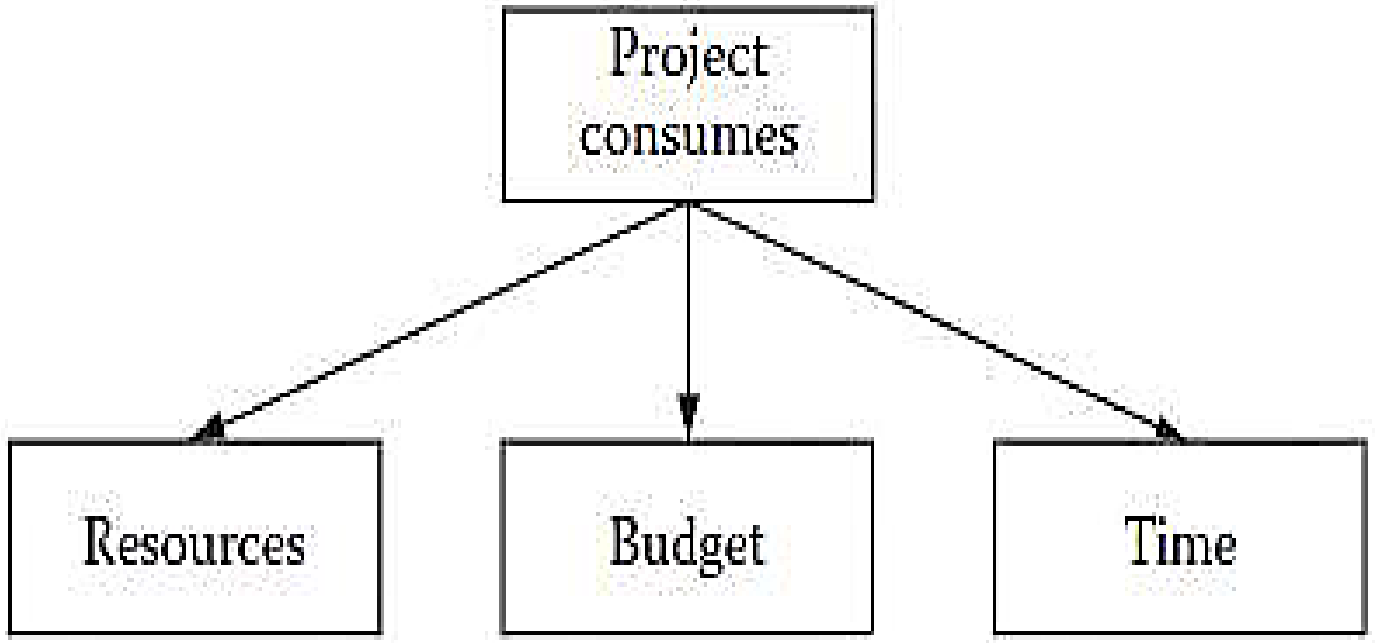
- The term “Software Project Management” contain the three words “Software”, “Project” & “Management”.

Software

- Software is the program and all associated documentation and configuration data which is needed to make these programs operate correctly.
- Software can be of different types: System software, Application software, Engineering/Scientific software, Embedded software, Product-line software, Web-applications, Artificial Intelligence software, etc.

Project

- A project is a set of related tasks that are coordinated to achieve a specific objective in a given time limit.
- A Project is a group of activities which has a clearly defined start and end time, a set of tasks, and a budget, that is developed to solve a well-defined goal or objective.



Management

- Management is the process of coordinating people and other resources to achieve the goals of the organization. It is the practice of executing and controlling the projects.
- Basically, the management involves the following activities:
    1. Planning- deciding what is to be done
    2. Organizing- making arrangements
    3. Staffing- selecting the right people for the job
    4. Directing- giving instructions
    5. Monitoring- checking on progress
    6. Controlling- taking action to remedy hold-ups
    7. Innovating- coming up with new solutions
    8. Representing- liaising with users, etc.

Project Management

- Project management is the discipline of initiating, planning, executing, controlling, and closing the work of a team to achieve specific goals and meet specific success criteria.
- The primary constraints are scope, time, quality and budget (cost).
- The secondary challenge is to optimize the allocation of necessary inputs and integrate them to meet pre-defined objectives.

✓Projects must be delivered on time.

✓Projects must be within cost.

✓Projects must be within scope.

✓Projects must meet customer quality requirements.

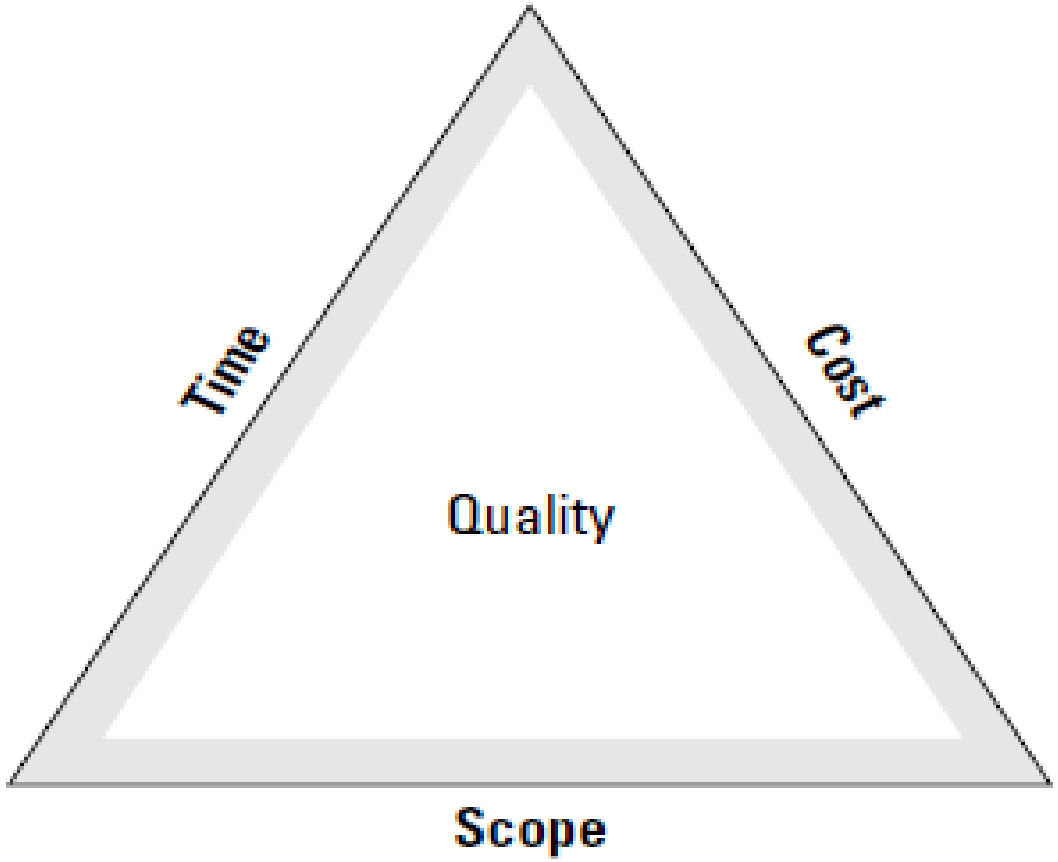


Figure: Project Management Triangle

Software Project

- A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.
- A software development project on the other hand is making software design based on customer requirements and implementing it into source code.
- This source code is then tested to make sure that it is defect free so that end users can use the software system without running into many problems.
- In software maintenance project, an already existing software product is modified to remove software defects, add new functionality, port the software product on some other operating system, etc.
- Software development and software maintenance projects together are referred to as software projects

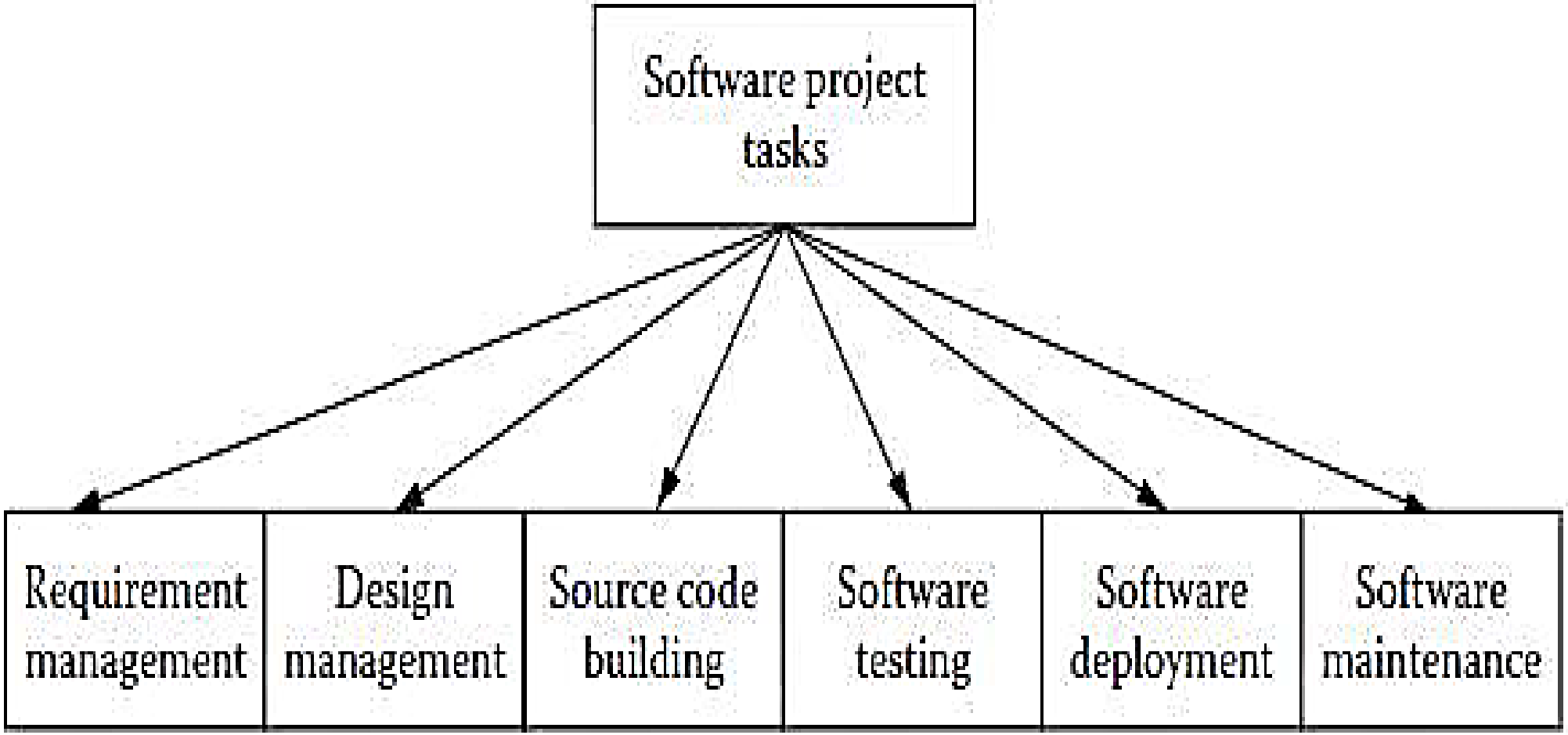
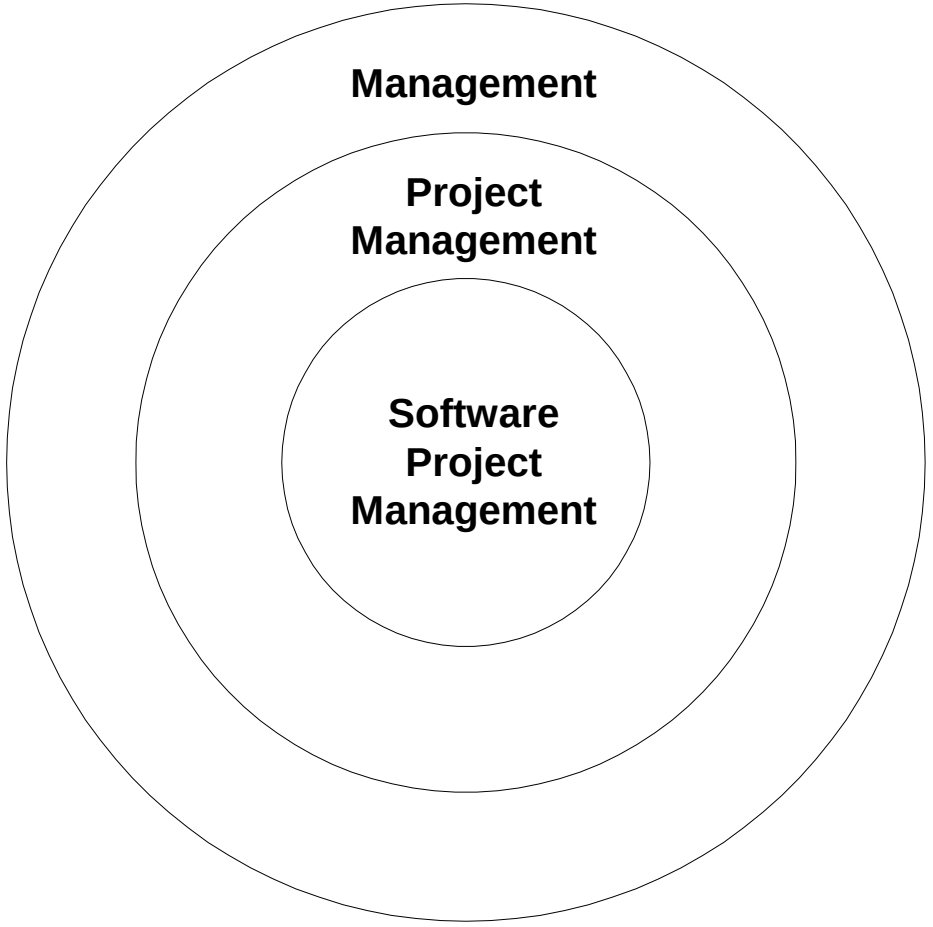


Figure: Tasks in Software Project

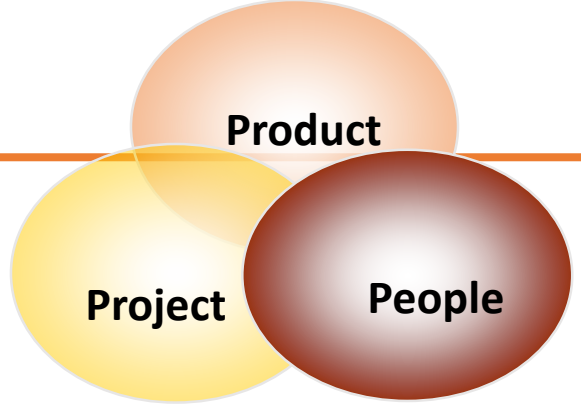
Software Project Management



“Software project management is a sub-discipline of project management in which software projects are planned, monitored and controlled.” - IEEE

Software Project Management

- Software project management is the art and science of planning and leading software projects.
- Software project management encompasses the knowledge, techniques, and tools necessary to manage the development of software products.
- A goal of any software project management is to develop/maintain a software product by applying good project management principles as well as software engineering principles so that the software project is delivered at minimum cost, within minimum time, and with good product quality.
- Software project management processes may include project initiation, project planning, project monitoring and control, and finally project closure.
- The software engineering processes may include requirement development, software design, software construction, software testing, and software maintenance.



Software

Product

1. Assessing Processes
2. Awareness of Process Standards
3. Defining the Product
4. Evaluating Alternative Processes
5. Managing Requirements
6. Managing Subcontractors
7. Performing the Initial Assessment
8. Selecting Methods and Tools
9. Tailoring Processes
10. Tracking Product Quality
11. Understanding Development Activities

Project

Project

12. Building a WBS
13. Documenting Plans
14. Estimating Costs
15. Estimating Effort
16. Managing Risks
17. Monitoring Development
18. Scheduling
19. Selecting Metrics
20. Selecting Project Mgmt. Tools
21. Tracking Process
22. Tracking Project Progress

Management

People

23. Appraising Performance
24. Handling Intellectual Property
25. Holding Effective Meetings
26. Interaction and Communication
27. Leadership
28. Managing Change
29. Negotiating Successfully
30. Planning Careers
31. Presenting Effectively
32. Recruiting
33. Selecting a Team
34. Teambuilding

34 Competencies Every Software Project Manager Needs to Know

1.1 Conventional Software Management Theory and Practice

Conventional Software Management

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

1. The best thing about software is its flexibility:

- It can be programmed to do almost anything.

2. The worst thing about software is its flexibility:

- The “almost anything” characteristic has made it difficult to plan, monitor, and control software development.

Conventional Software

In the mid-1990s, three important analyses were performed on the software engineering industry.

All three analyses given the same general conclusion: "The success rate for software projects is very low".

They Summarized as follows:

1) Software development is still highly unpredictable. Only 10% of software projects are delivered successfully within initial budget and scheduled time.
2) Management discipline is more differentiator in success or failure than are technology advances.
3) The level of software scrap and rework is indicative of an immature process.

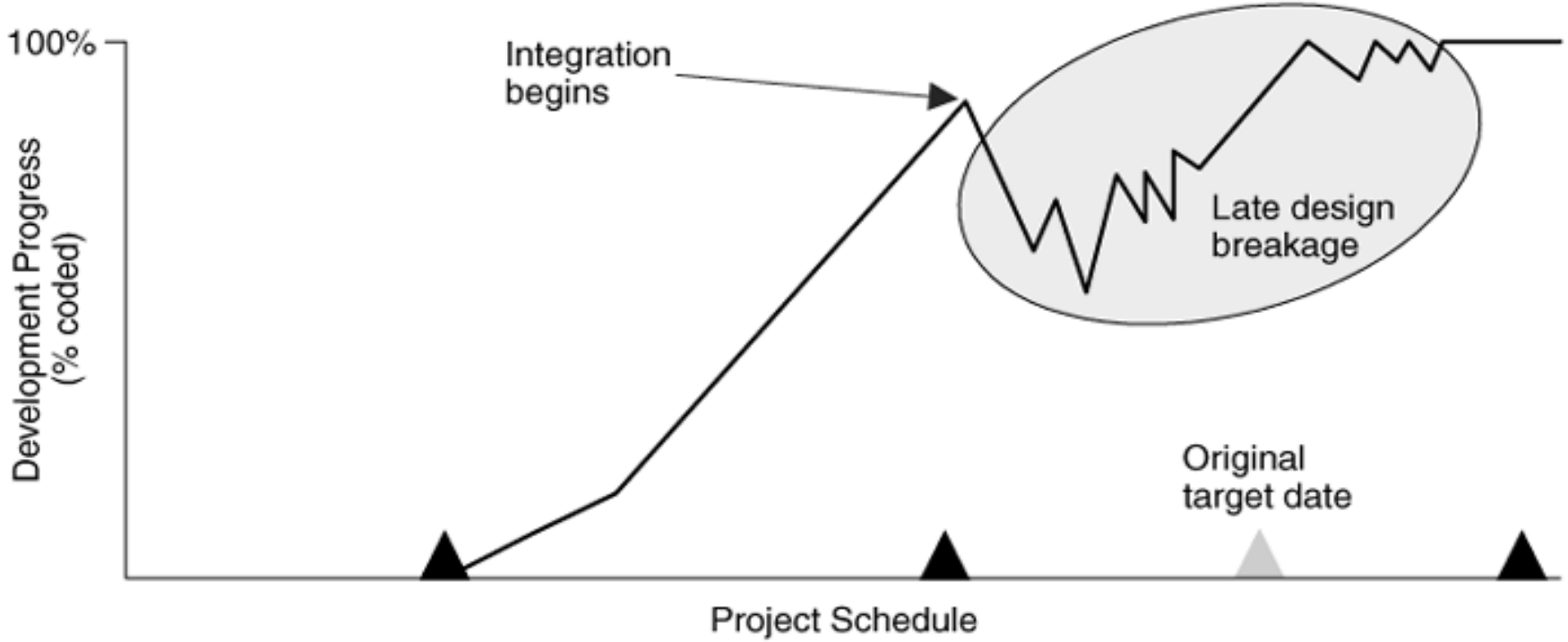
Conventional Software

- Conventional software development process today is unreliable and immature.
- The Waterfall Method needs improvements to work satisfactorily.
- Conventional practices provide a benchmark for future improvements and changes to software development methods.

Typical Progress Profile – Conventional Project

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

Sequential activities: requirements — design — coding — integration — testing



The Software Development Experience

- Development is highly unpredictable.
  - Only 10% of projects on time and budget.
- 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.

Classical Waterfall Model

It is the baseline process for most conventional software projects have used.
We can examine this model in two ways:

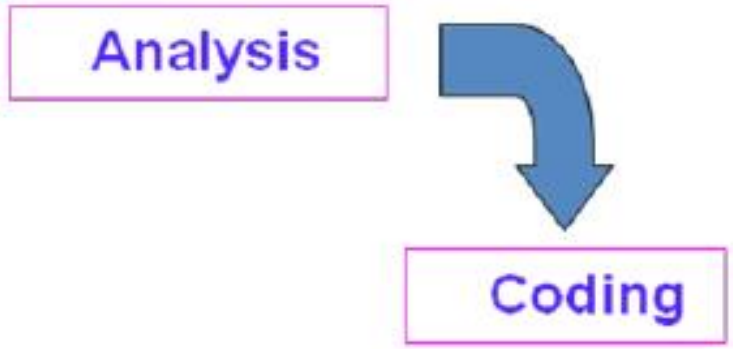
- IN THEORY
- IN PRACTICE

IN THEORY:

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:

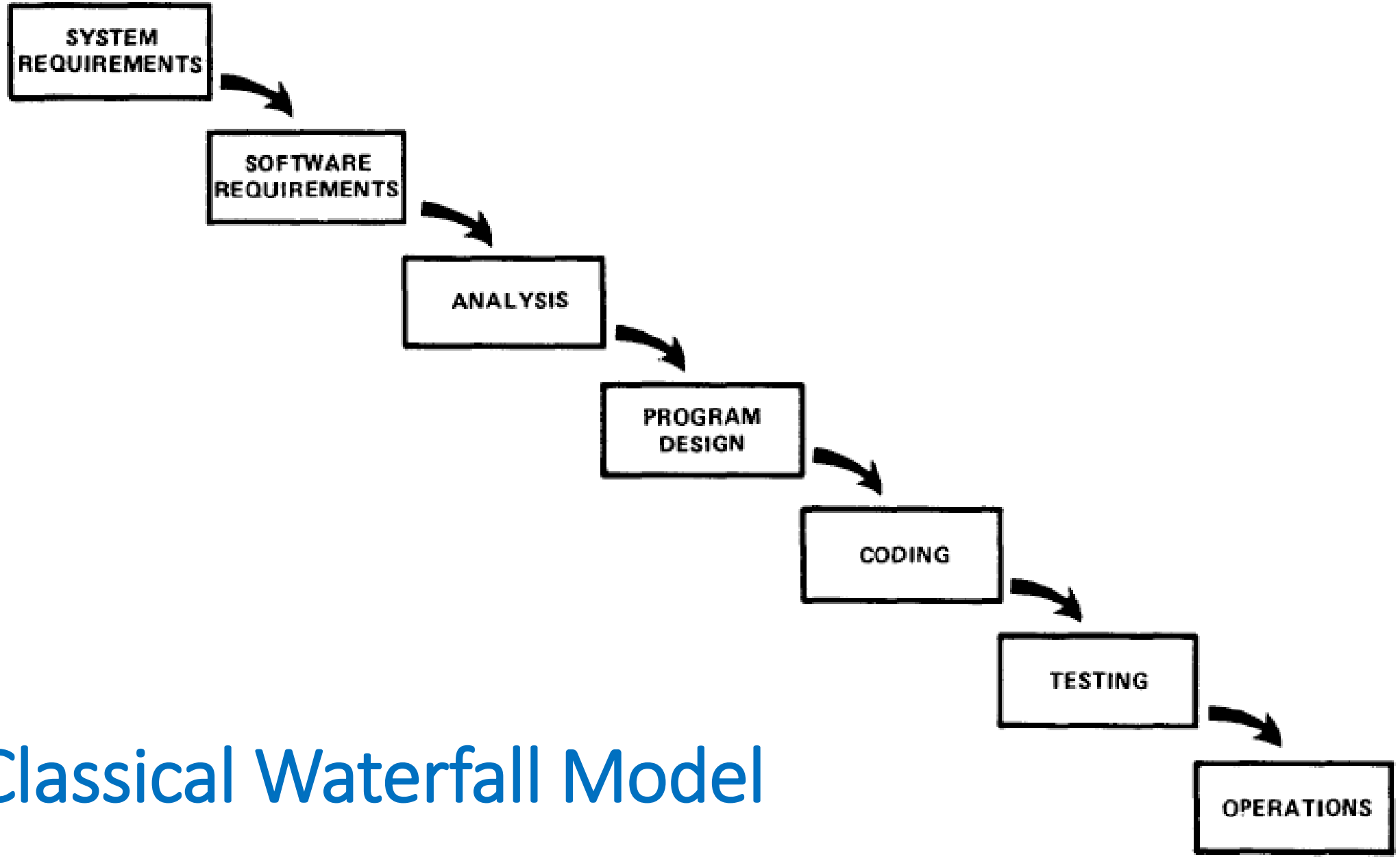
1.There are two essential steps common to the development of computer programs:

- analysis
- coding



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

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



Classical Waterfall Model

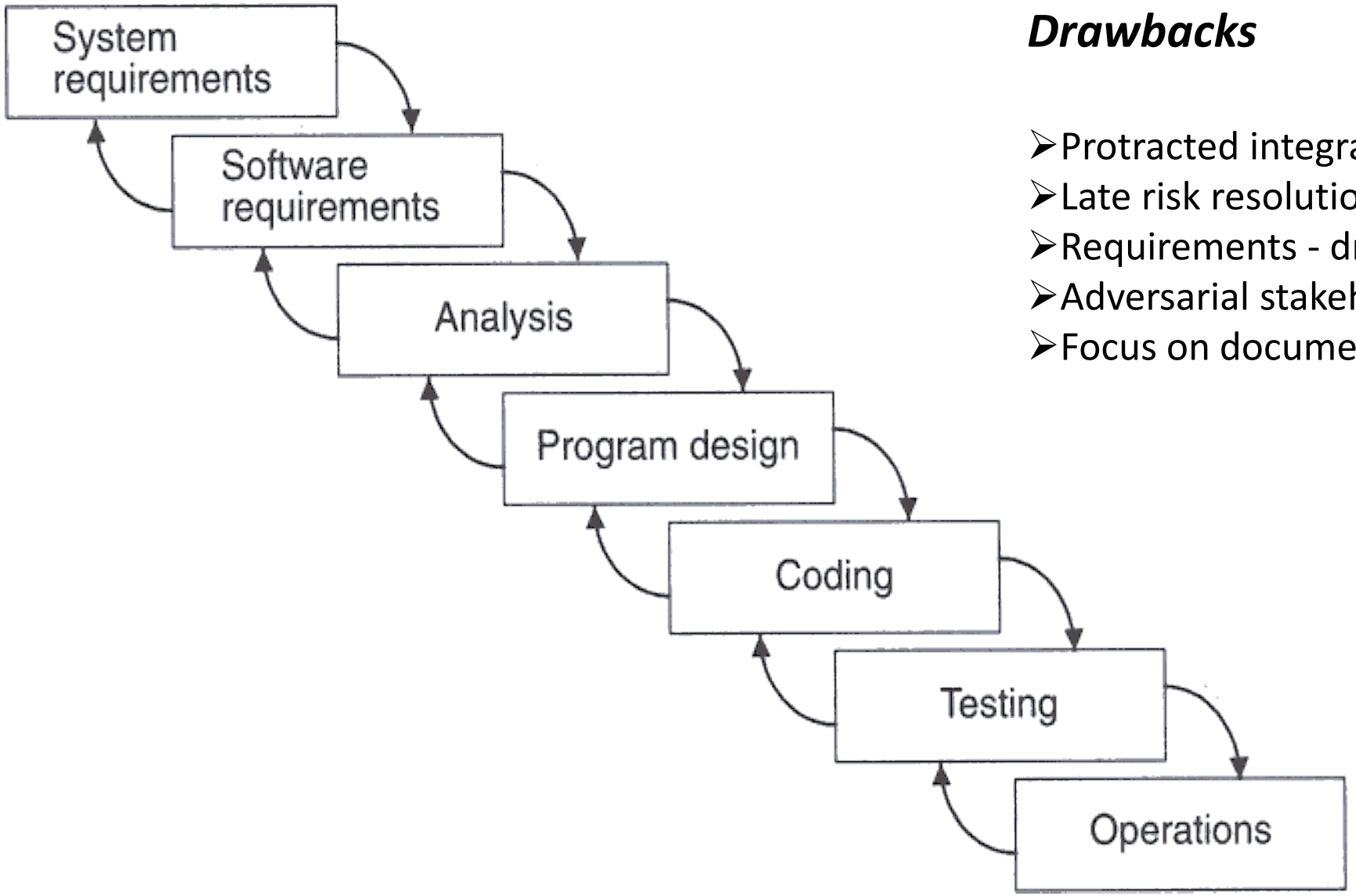
Waterfall Model

- Inflexible partitioning of the project into distinct stages
- This makes it difficult to respond to changing customer requirements
- This model is only appropriate when:
  - The requirements are well-understood and/or
  - The level of formality is high (e.g. it is essential to "freeze" the requirement document)

2. In order to mange and control all of the intellectual freedom associated with software development one should follow the following steps:

- **System requirements definition**
- **Software requirements definition**
- **Program design and testing**

These steps addition to the analysis and coding steps



Drawbacks

- Protracted integration and late design breakage
- Late risk resolution
- Requirements - driven functional decomposition
- Adversarial stakeholder relationships
- Focus on document and review meetings

3. Since the testing phase is at the end of the development cycle in the waterfall model, it may be risky and invites failure. So we need to do either the requirements must be modified or a substantial design changes is warranted by breaking the software in to different pieces.

There are five improvements to the basic waterfall model that would eliminate most of the development risks are as follows:

a) Complete program design before analysis and coding begin (program design comes first):

- By this technique, the program designer give surety that the software will not fail because of storage, timing, and data fluctuations.
- Begin the design process with program designer, not the analyst or programmers.
- 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) Maintain current and complete documentation (Document the design):

- It is necessary to provide a lot of documentation on most software programs.
- Due to this, helps to support later modifications by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

c) Do the job twice, if possible (Do it twice):

- If a computer program is developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations are concerned.
- "Do it N times" approach is the principle of modern-day iterative development.

d) Plan, control, and monitor testing:

- The biggest user of project resources is the test phase. This is the phase of greatest risk in terms of cost and schedule.
- In order to carryout proper testing the following things to be done:
  - i) Employ a team of test specialists who were not responsible for the original design.
  - ii) Employ visual inspections to spot the obvious errors like dropped minus signs, missing factors of two, jumps to wrong addresses.
  - iii) Test every logic phase.
  - iv) Employ the final checkout on the target computer.

e) Involve the customer:

- It is important to involve the customer in a formal way so that he has committed himself at earlier points before final delivery by conducting some reviews such as,
  - i) Preliminary software review during preliminary program design step.
  - ii) Critical software review during program design.
  - iii) Final software acceptance review following testing.

IN PRACTICE:

- Whatever the advices that are given by the software developers and the theory behind the waterfall model, some software projects still practice the conventional software management approach.

Projects intended for trouble frequently exhibit the following symptoms:

i) Protracted (delayed) integration

- ❑ 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.
- ❑ Here the testing activities consume 40% or more life-cycle resources.

Expenditures – Conventional Project

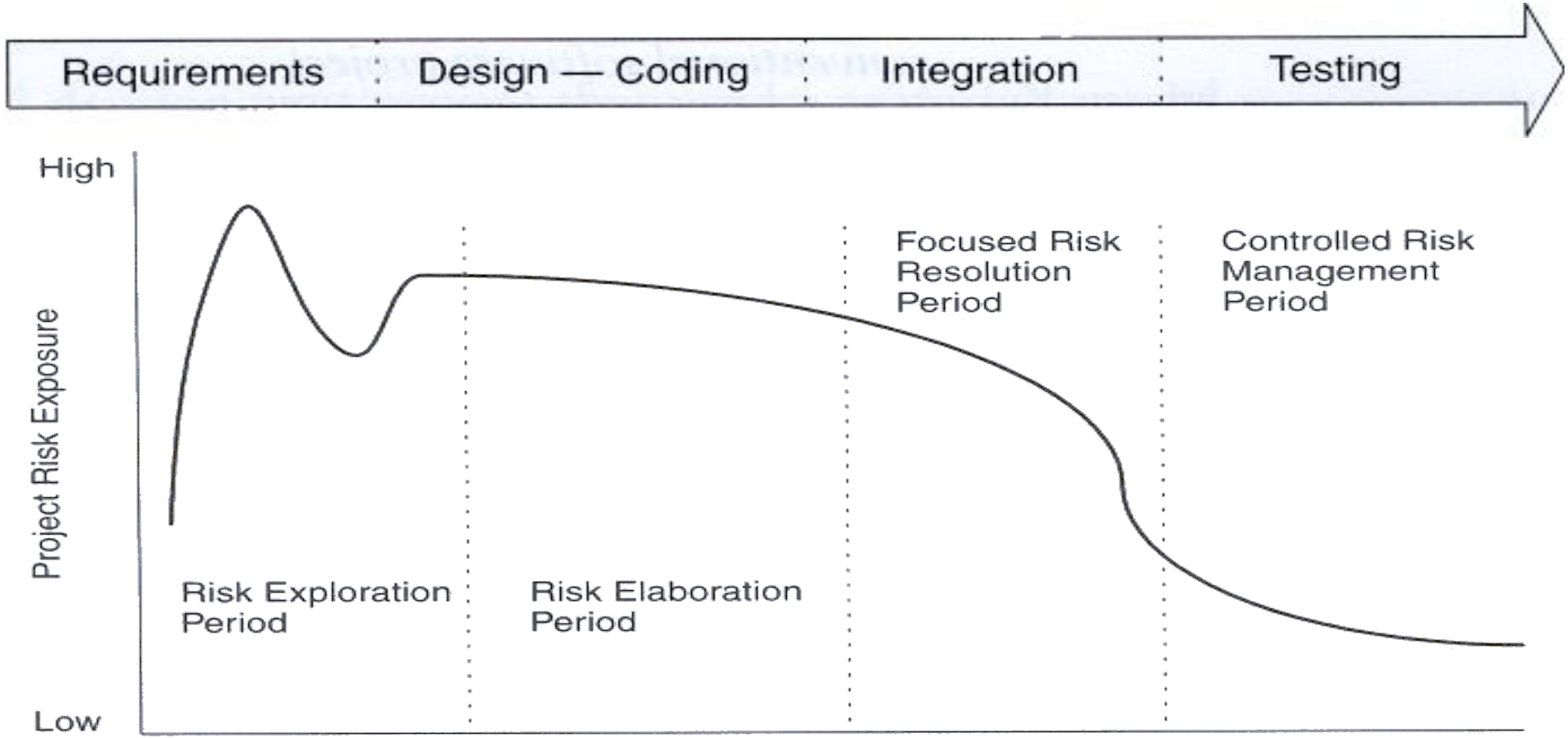
• Management	5%
• Requirements	5%
• Design	10 %
• Code and unit testing	30%
• Integration and test	40%
• Deployment	5%
• Environment	5%

ii) Late Risk Resolution

- A serious issues associated with the waterfall life cycle was the lack of early risk resolution.
- The risk profile of a waterfall model is shown in the figure.
- It includes four distinct periods of risk exposure, where risk is defined as “the probability of missing a cost, schedule, feature, or quality goal”.

What happens in practice?

Risk Profile – Conventional Project



iii) Requirements-Driven Functional Decomposition

- Traditionally, the software development process has been requirement-driven: An attempt is made to provide a precise requirements definition and then to implement exactly those requirements.
- This approach depends on specifying requirements completely and clearly before other development activities begin.
- It frankly treats all requirements as equally important.
- Specification of requirements is a difficult and important part of the software development process.

iv) Adversarial Stakeholder Relationships

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

- The contractor prepared a draft contact-deliverable document that captured an intermediate artifact and delivered it to the customer for approval.
- The customer was expected to provide comments (within 15 to 30 days)
- The contractor integrated these comments and submitted a final version for approval (within 15 to 30 days)

Typical Software Project Documents and Control

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

2. The customer was expected to provide comments {typically within 15 to 30 days).

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

v) Focus on Documents and Review Meetings

- The conventional process focused on various documents that attempted to describe the software product.
- Contractors produce literally tons of paper to meet milestones and demonstrate progress to stakeholders, rather than spend their energy on tasks that would reduce risk and produce quality software.
- Most design reviews resulted in low engineering and high cost in terms of the effort and schedule involved in their preparation and conduct.

Typical Program Experience

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

Summary of Problems in Conventional Practice……

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

Five Improvements for the Waterfall Model to Work

- ❑Complete program design before analysis and coding begin.
- ❑Maintain current and complete documentation.
- ❑Do the job twice, if possible.
- ❑Plan, control, and monitor testing.
- ❑Involve the customer.

Conventional Software Management Performance

Boehm's Top Ten List

Barry Boehm's Top 10 "Industrial Software Metrics":

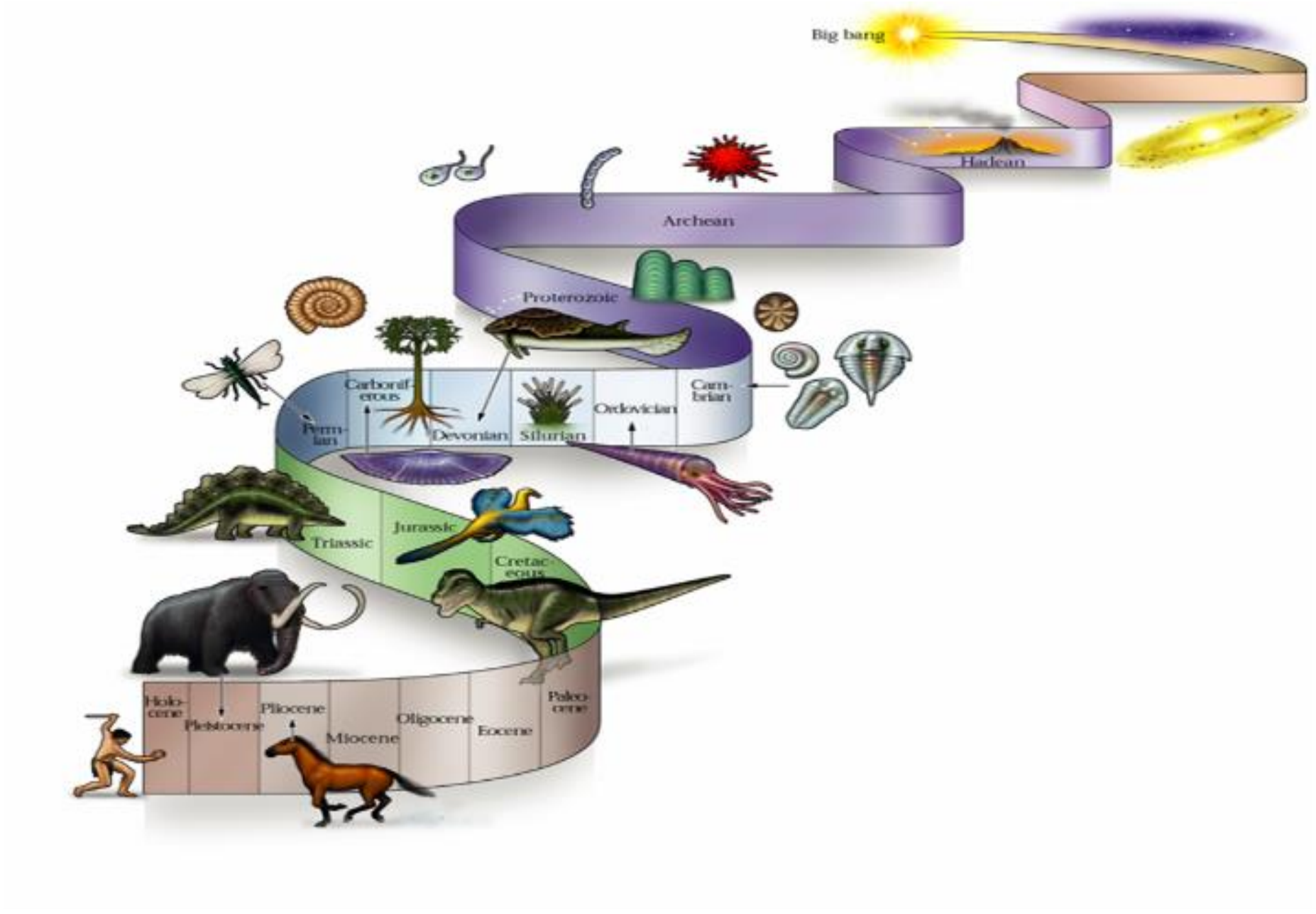
- Fixing after delivery costs 100 times as much as early fix.
- You can compress schedule 25%, but no more.
- For every $1 spent on development you will spend $2 on maintenance.
- Costs are primarily a function of source lines of code.
- Variations among people account for the biggest differences in productivity
- Ratio of software to hardware cost is 85:15 and still growing.
- Only about 15% of software development cost is due to programming.
- Software systems cost 3 times as much as software programs. Software system products cost 9 times as much.
- Walkthroughs catch 60% of the errors.
- 80% of the contribution comes from 20% of the contributors.

The 80/20 Rule

- 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 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.

1.2 Software Economics and Cost Estimations

Evolution of Software Economics



Evolution of Software Economics

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

- **Size:** Which is measured in terms of the number of Source Lines Of Code or the number of function points required to develop the required functionality.
- **Process:** Used to produce the end product, in particular the ability of the process is to avoid non value-adding activities (rework, bureaucratic delays, communications overhead).
- **Personnel:** The capabilities of software engineering personnel, and particularly their experience with the computer science issues and the application domain issues of the project.
- **Environment:** Which is made up of the tools and techniques available to support efficient software development and to automate the process.
- **Quality:** It includes its features, performance, reliability, and flexibility.

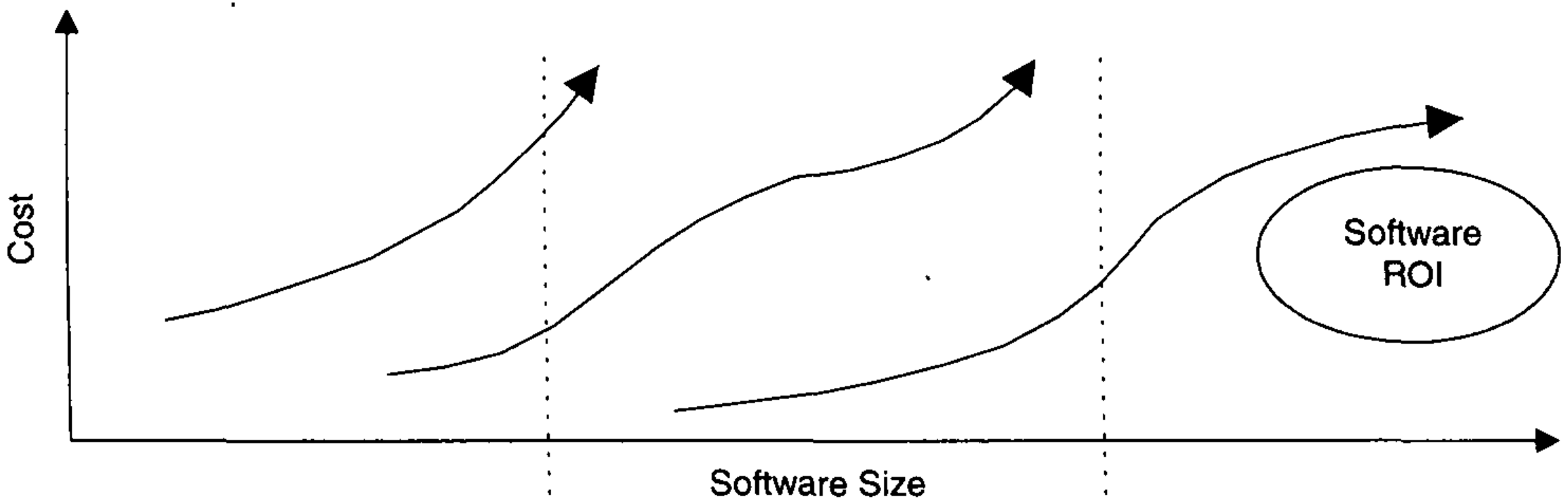
The relationship among these parameters and estimated cost can be calculated by using,

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

- One important aspect of software economics is that the relationship between effort and size exhibits a diseconomy of scale and is the result of the process exponent being greater than 1.0.
- Converse to most manufacturing processes, the more software you build, the more expensive it is per unit item.
- There are three generations of basic technology advancement in tools, components, and processes are available.

1. **Conventional:** 1960 and 1970, Craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable.
2. **Transition:** 1980 and 1990, 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 like, OS, DBMS, Networking and GUI
3. **Modern practices:** 2000 and later, software production.
   - 70% component-based,
   - 30% custom

Target objective: improved ROI



Conventional	Transition	Modern Practices
- 1960s–1970s - Waterfall model - Functional design - Diseconomy of scale	- 1980s–1990s - Process impovement - Encapsulation-based - Diseconomy of scale	- 2000 and on - Iterative development - Component-based - Return on investment

Corresponding environment, size, and process technologies

Conventional	Transition	Modern Practices
Environments/tools: Custom	**Environment/tools:** Off-the-shelf, separate	**Environment/tools:** Off-the-shelf, integrated
Size: 100% custom	**Size:** 30% component-based 70% custom	**Size:** 70% component-based 30% custom
Process: Ad hoc	**Process:** Repeatable	**Process:** Managed/measured

Conventional	Transition	Modern Practices
Typical project performance		
Predictably bad	**Unpredictable**	**Predictable**
Always:	Infrequently:	Usually:
Over budget	On budget	On budget
Over schedule	On schedule	On schedule

Figure: *Three generations of software economics leading to the target objective*

What Does *Return On Investment - ROI* Mean?

A performance measure used to evaluate the efficiency of an investment or to compare the efficiency of a number of different investments. To calculate ROI, the benefit (return) of an investment is divided by the cost of the investment; the result is expressed as a percentage or a ratio.

The return on investment formula:

$$\text{ROI} = \frac{(\text{Gain from Investment} - \text{Cost of Investment})}{\text{Cost of Investment}}$$

Return on investment is a very popular metric because of its versatility and simplicity. That is, if an investment does not have a positive ROI, or if there are other opportunities with a higher ROI, then the investment should be not be undertaken.

Project Sizes:

- Size as team strength could be :
  - Trivial (Minor) Size: 1 person
  - Small Size: 5 people
  - Moderate Size: 25 people
  - Large Size: 125 people
  - Huge Size: 625 people
- The more the size, the greater are the costs of management overhead, communication, synchronizations among various projects or modules, etc.

Reduce Software Size:

- The less software we write, the better it is for project management and for product quality
- The cost of software is not just in the cost of ‘coding’ alone; it is also in
  - Analysis of requirements
  - Design
  - Review of requirements, design and code
  - Test Planning and preparation
  - Testing
  - Bug fix
  - Regression testing
- ‘Coding’ takes around 15% of development cost
- Clearly, if we reduce 15 hrs of coding, we can directly reduce 100 hrs of development effort, and also reduce the project team size appropriately !
- Size reduction is defined in terms of human-generated source code. Most often, this might still mean that the computer-generated executable code is at least the same or even more
- Software Size could be reduced by
  - Software Re-use
  - Use of COTS (Commercial Off-The Shelf Software)
  - Programming Languages

Pragmatic software cost estimation

- If there is no proper well-documented case studies then it is difficult to estimate the cost of the software. It is one of the critical problem in software cost estimation.
- But the cost model vendors claim that their tools are well suitable for estimating iterative development projects.

- In order to estimate the cost of a project the following three topics should be considered,
  1) Which cost estimation model to use.
  2) Whether to measure software size in SLOC or function point.
  3) What constitutes a good estimate.

- There are a lot of software cost estimation models are available such as:
  COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20.

- Of which COCOMO is one of the most open and well-documented cost estimation models

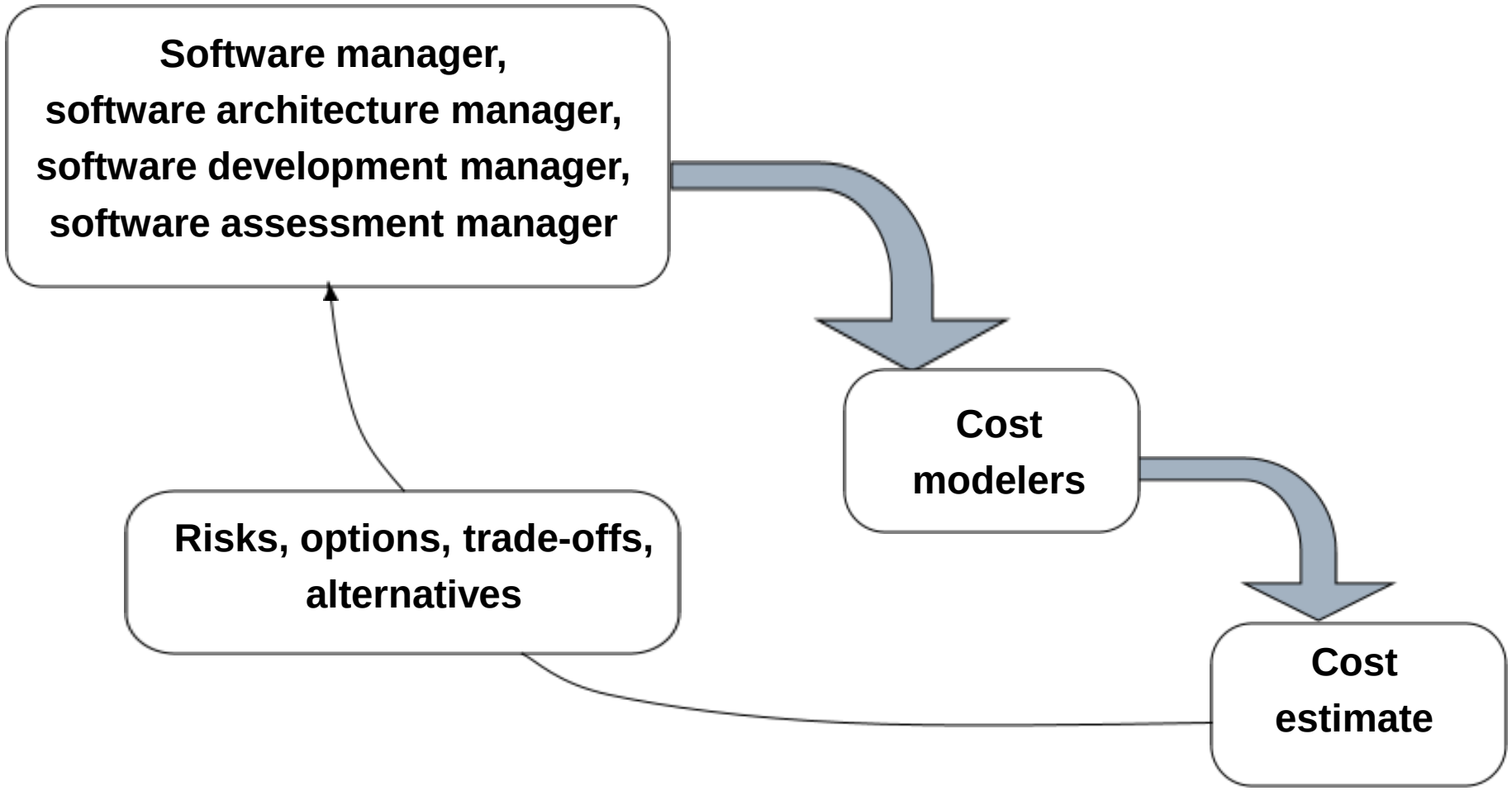
Pragmatic software cost estimation

- The software size can be measured by using
  1) SLOC
  2) Function points
- Most software experts argued that the SLOC is a poor measure of size. But it has some value in the software Industry.
- SLOC worked well in applications that were custom built why because of easy to automate and instrument.
- Now a days there are so many automatic source code generators are available and there are so many advanced higher-level languages are available. So SLOC is a uncertain measure.
- The main advantage of function points is that this method is independent of the technology and is therefore a much better primitive unit for comparisons among projects and organizations.
- The main disadvantage of function points is that the primitive definitions are abstract and measurements are not easily derived directly from the evolving artifacts.
- Function points is more accurate estimator in the early phases of a project life cycle. In later phases, SLOC becomes a more useful and precise measurement basis of various metrics perspectives.
- The most real-world use of cost models is bottom-up rather than top-down.
- The software project manager defines the target cost of the software, then manipulates the parameters and sizing until the target cost can be justified.

Pragmatic software cost estimation

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

The predominant cost estimation process



1.3 Improving Software Economics

Parameters of the Software Cost Model

Five basic parameters of the software cost model:

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

Important Trends in Improving Software Economics

Cost model parameters Trends

Size Abstraction and component based development technologies	→	**Higher order languages** (C++, 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, etc.) **Open systems** **Hardware platform performance** **Automation of coding, documents, testing, analysis**
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 is usually to produce a product that achieves the design goals with the minimum amount of human-generated source material."

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

Reducing Software Product Size - Languages

UFP -Universal Function Points
The basic units of the function points are external user inputs, external outputs, internal logic data groups, external data interfaces, and external inquiries.

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

SLOC metrics
They are useful estimators for software after a candidate solution is formulated
And an implementation language is known.

Reducing Software Product Size – Object-Oriented Methods

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

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

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

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

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

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

Reducing Software Product Size – Reuse

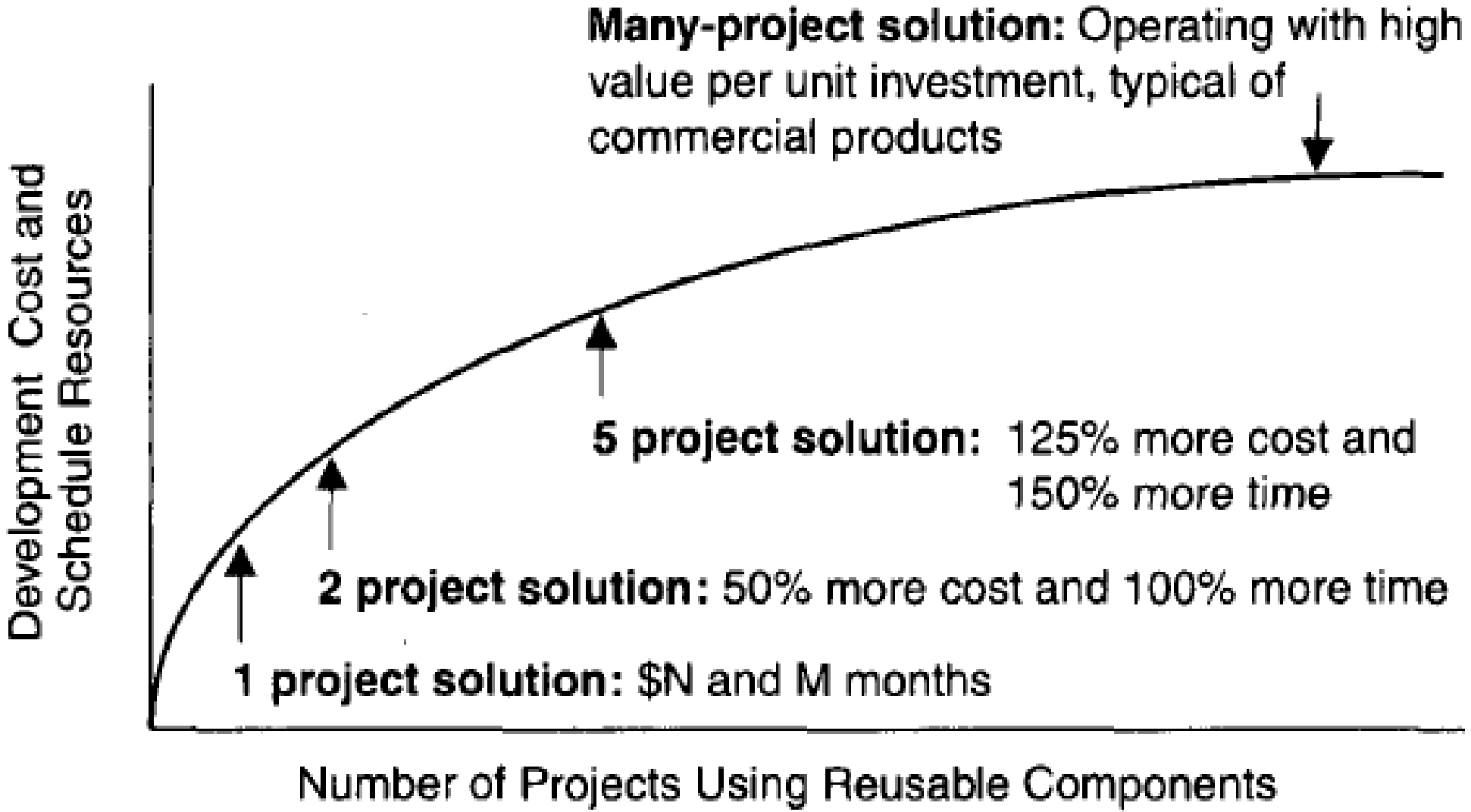


FIGURE 3-1. *Cost and schedule investments necessary to achieve reusable components*

Reducing Software Product Size – Commercial Components

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

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

1.4 Software Process

Improving Software Processes

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 budget, schedule, quality
Audience	Acquisition authorities, customers Organizational management	Software project managers Software engineers	Subproject managers Software engineers
Metrics	Project predictability Revenue, market share	On budget, on schedule Major milestone success Project scrap and rework	On budget, on schedule Major milestone progress Release/iteration scrap and rework
Concerns	Bureaucracy vs. standardization	Quality vs. financial performance	Content vs. schedule
Time scales	6 to 12 months	1 to many years	1 to 6 months

Three levels of processes and their attributes

1.5 Team Effectiveness, Software Environment, and Quality Target

Improving Team Effectiveness

- ❑ Teamwork is much more important than the sum of the individuals.
- ❑ With software teams, a project manager needs to configure a balance of solid talent with highly skilled people in the leverage positions.

Some maxims of team management include the following:

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

Improving Team Effectiveness

Boehm Five Staffing Principles:

- The principle of top talent: *Use better and fewer people.*
- The principle of job matching: *Fit the task to the skills an motivation of the people available.*
- The principle of career progression: *An organization does best in the long run by helping its people to self-actualize*.
- The principle of team balance: *Select people who will complement and harmonize with one another.*
- The principle of phase-out: *Keeping a misfit on the team doesn't benefit anyone.*

Improving Team Effectiveness

Important Project Manager Skills:

- Hiring skills. Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.
- Customer-interface skill. Avoiding adversarial relationships among stake-holders is a prerequisite for success.
- Decision-making skill. The jillion books written about management have failed to provide a clear definition of this attribute. We all know a good leader when we run into one, and decision-making skill seems obvious despite its intangible definition.
- Team-building skill. Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.
- Selling skill. Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives. In practice, selling requires continuous negotiation, compromise, and empathy.

Achieving Required Quality

Key practices that improve overall software quality:

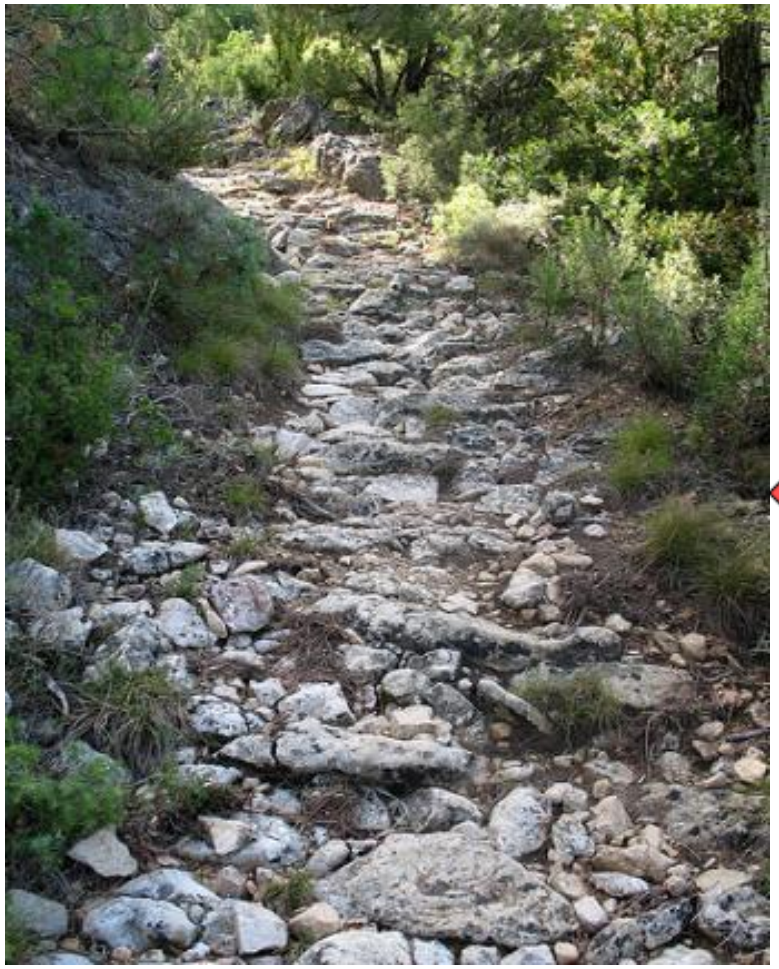
- Focusing on driving requirements and critical use cases early in the life cycle, focusing on requirements completeness and traceability late in the life cycle, and focusing throughout the life cycle on a balance between requirements evolution, design evolution, and plan evolution
- Using metrics and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

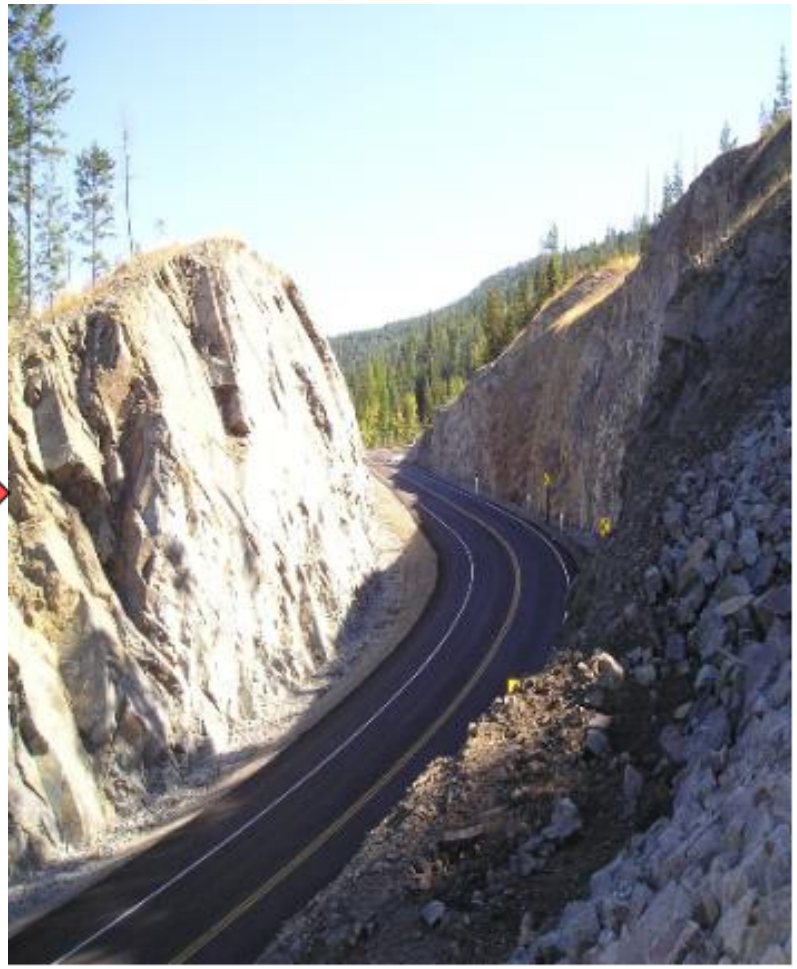
TABLE 3-5. *General quality improvements with a modern process*

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

1.6 Principles of Conventional Software Engineering

The Old Way and the New





The Principles of Conventional Software Engineering

1. **Make quality #1.** Quality must be quantified and mechanism 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.
5. **Evaluate design alternatives**. After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use an "architecture" simply because it was used in the requirements specification.

The Principles of Conventional Software Engineering

6. **Use an appropriate process model.** Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases.** Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation through-out the life cycle. Why should software engineers use Ada for requirements, design, and code unless Ada were optimal for all these phases?
8. **Minimize intellectual distance.** To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure.
9. **Put techniques before tools.** An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer.
10. **Get it right before you make it faster.** It is far easier to make a working program run than it is to make a fast program work. Don't worry about optimization during initial coding.

The Principles of Conventional Software Engineering

11. **Inspect code.** Inspecting the detailed design and code is a much better way to find errors than testing.
12. **Good management is more important than good technology.** The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success.** Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process will succeed. The wrong people with appropriate tools, languages, and process will probably fail.
14. **Follow with care.** Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping-all these might increase quality, decrease cost, and increase user satisfaction. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal.
15. **Take responsibility.** When a bridge collapses we ask, "what did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant design.

The Principles of Conventional Software Engineering

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.

The Principles of Conventional Software Engineering

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 become more and more disorganized.
29. **People and time are not interchangeable.** Measuring a project solely by person-months makes little sense.
30. **Expert excellence.** Your employees will do much better if you have high expectations for them.

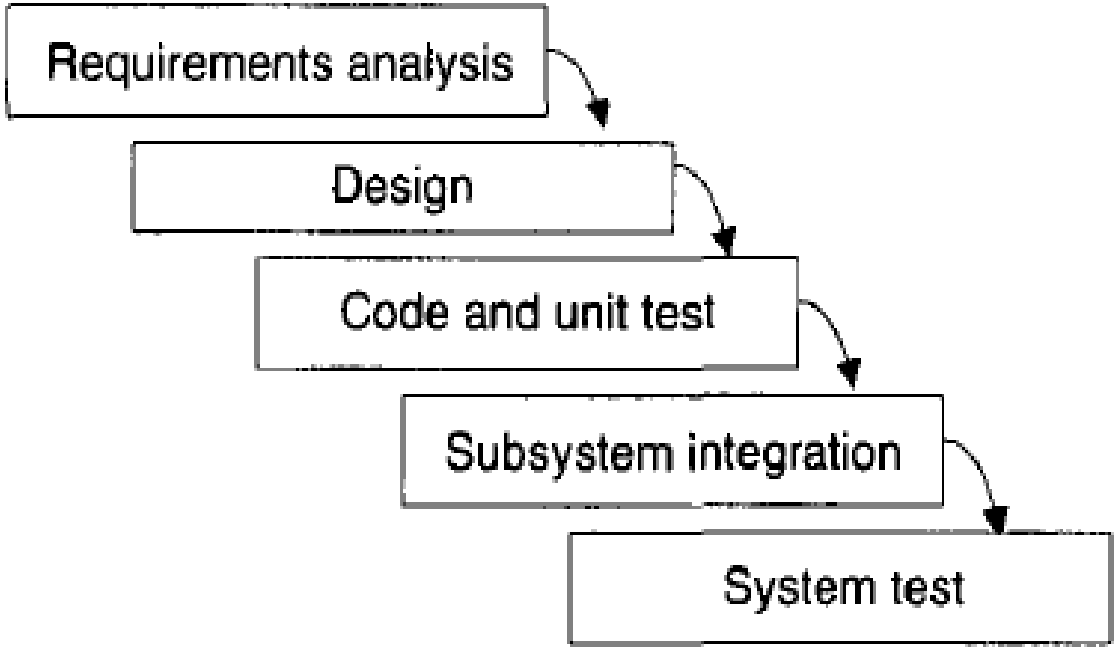
1.7 Principles of Modern Software Management

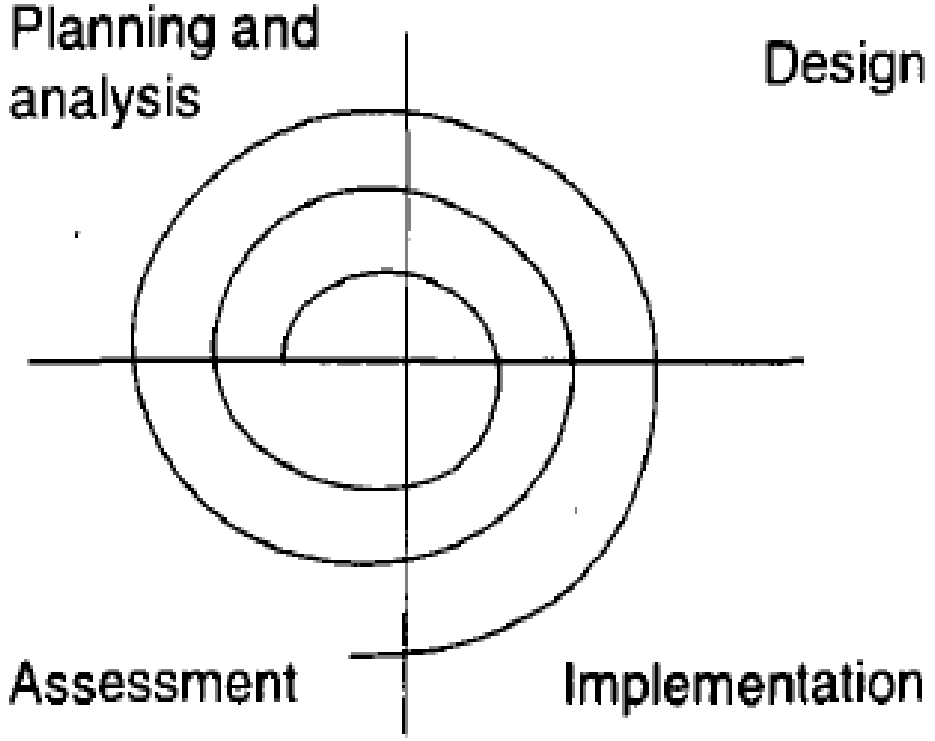
Waterfall Process

Requirements first
Custom development
Change avoidance
Ad hoc tools

Iterative Process

Architecture first
Component-based development
Change management
Round-trip engineering





The Principles of Modern Software Management

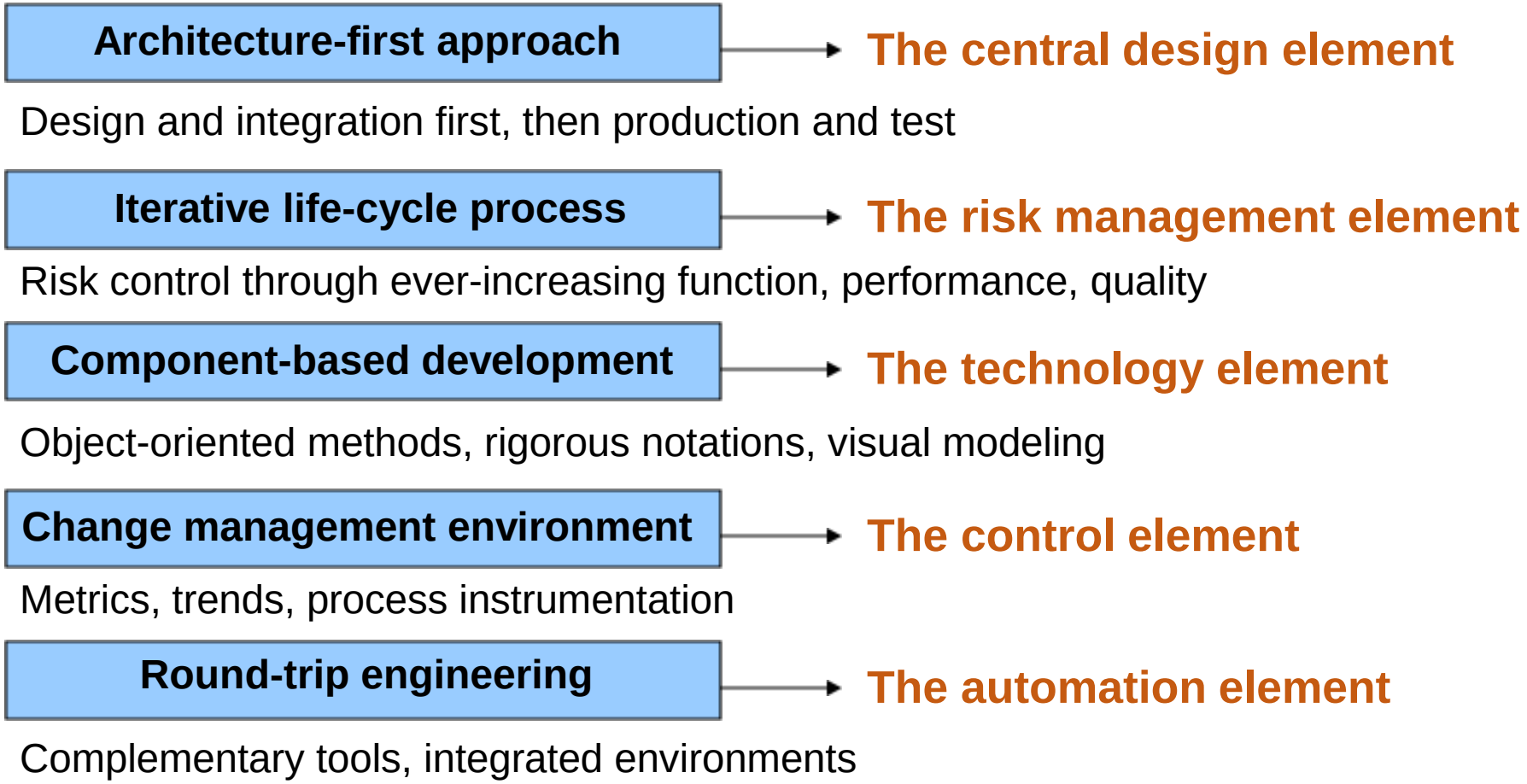


Figure: The top 5 principles of a modern process

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

CONVENTIONAL PROCESS: TOP 10 RISKS	IMPACT	MODERN PROCESS: INHERENT RISK RESOLUTION FEATURES
1. Late breakage and excessive scrap/rework	Quality, cost, schedule	Architecture-first approach Iterative development Automated change management Risk-confronting process
2. Attrition of key personnel	Quality, cost, schedule	Successful, early iterations Trustworthy management and planning
3. Inadequate development resources	Cost, schedule	Environments as first-class artifacts of the process Industrial-strength, integrated environments Model-based engineering artifacts Round-trip engineering
4. Adversarial stakeholders	Cost, schedule	Demonstration-based review Use-case-oriented requirements/testing
5. Necessary technology insertion	Cost, schedule	Architecture-first approach Component-based development

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

CONVENTIONAL PROCESS: TOP 10 RISKS	IMPACT	MODERN PROCESS: INHERENT RISK RESOLUTION FEATURES
6. Requirements creep	Cost, schedule	Iterative development Use case modeling Demonstration-based review
7. Analysis paralysis	Schedule	Demonstration-based review Use-case-oriented requirements/testing
8. Inadequate performance	Quality	Demonstration-based performance assessment Early architecture performance feedback
9. Overemphasis on artifacts	Schedule	Demonstration-based assessment Objective quality control
10. Inadequate function	Quality	Iterative development Early prototypes, incremental releases

1.8 Iterative Process

Application precedentedness.

Process flexibility.

Architecture risk resolution.

Team cohesion.

Software process maturity.

Unit 1: Important Questions

1.	Explain briefly Waterfall model. Also explain Conventional s/w management performance?
2.	Define Software Economics. Also explain Pragmatic s/w cost estimation?
3.	Explain Important trends in improving Software economics?
4.	Explain five staffing principal offered by Boehm. Also explain Peer Inspections?
5..	Explain principles of conventional software engineering?
6.	Explain briefly principles of modern software management


Any Questions
?

Book References:

1. Software Project Management – A Unifies Framework, Walker Royce, 1998, Addison Wesley
2. Software Project Management – From Concept to Deployment, Conway, K., 2001.
3. Software Project Management, Bob Hughes and Mike Cotterell, Latest Publication
4. Software Project Management, Rajeev Chopra, 2009
5. Software Engineering – A Practitioner's approach, Roger S. Pressman Latest Plublication
6. Managing Global Software Projects, Ramesh, 2001, TMH


Thank You