Software Systems Development 10

Software Project Organization and Management

- **Prerequisite**: SSD9 Software Specification, Testing, and Maintenance (may be taken concurrently).
- **Course textbook**: Royce,Walker. *Software Project Management: A Unified Framework.* Reading, MA: Addison Wesley Longman, Inc., 1998. ISBN: 0-201-30958-0.

Organization

The book is laid out in five parts, each with multiple chapters:

- **Part I, Software Management Renaissance.** Describes the current state of software management practice and software economics, and introduces the state transitions necessary for improved software return on investment.
- **Part II, A Software Management Process Framework.** Describes the process primitives and a framework for modern software management, including the life-cycle phases, artifacts, workflows, and checkpoints.
- **Part III, Software Management Disciplines.** Summarizes some of the critical techniques associated with planning, controlling, and automating a modern software process.
- **Part IV, Looking Forward.** Hypothesizes the project performance expectations for modern projects and next-generation software economics, and discusses the culture shifts necessary for success.
- **Part V, Case Studies and Backup Material.** Five appendixes provide substantial foundations for some of the recommendations, guidance, and opinions presented elsewhere.

The purpose of SSD10

1. Learn the organizational and management aspects of software projects.
2. Learn project management techniques for scheduling, costing, risk analysis, and project organization.
3. Learn to examine and objectively critique various kinds of planning and management artifacts.
4. Learn to develop standard project management documents and supplementary artifacts.
5. Learn a modern framework for managing the software development process.
6. Learn to reason about software development models.
7. Learn principles concerning leadership, liability, intellectual property, confidentiality issues, and management of customer relationships.

Hold Positions as Software Project Managers

Those who certify in this course will be able to handle a wide range of responsibilities that complex software projects typically involve. These include:

a) development of iteration-based project plans and schedules,

b) cost estimation and effort allocation,

c) management of customer relations,

d) risk assessment and mitigation,

e) communication with top management,

f) production of standards-based project documentation, and

g) interfacing with a legal department to deal with issues involving confidentiality, intellectual property, patents, and copyrights.

Syllabus

Exercise 1 Thinking in the New Way

Exercise 2 Life-Cycle Phases

Exercise 3 Artifacts of the Process

Exercise 4 Model-Based Software Architectures

Exercise 5 Workflows of the Process

Exercise 6 Checkpoints of the Process

Exercise 7 Iterative Project Planning

Exercise 8 Project Organization and Responsibilities

Exercise 9 Process Automation

Exercise 10 Tools (Gantt, PERT, and Resource Charts)

Exercise 11 Project Control and Process Instrumentation

Exercise 12 Tailoring the Process

Exercise 13 Ethics

Periods	Tasks to complete		Total
1st attestation	Exercise 1, Exercise 2, Exercise 3, Exercise 4, Exercise 5, Exercise 6, Exercise 7, MCQ1, MCQ2, MCQ3, MCQ4, MCQ5, MCQ6, MCQ7, MCQ8, MCQ9, Exam1 MCQ, Exam1 PQ Quizzes Attendance	3 3 3 3 3 3 3 9 10 30 10 20	100
2nd attestation	Exercise 8, Exercise 9, Exercise 10, Exercise 11, Exercise 12, Exercise 13, MCQ10, MCQ11, MCQ12, MCQ13, MCQ14, MCQ15, MCQ16, MCQ17, MCQ18, Exam2 MCQ, Exam2 PQ, Exam3 MCQ, Exam3 PQ Quizzes Attendance	3 3 3 4 4 4 9 5 15 5 15 10 20	100
Final exam	**Certification Exam Multiple-Choice** **Certification Exam Practical**	12 88	100
Total	**0,3*1stAtt+0,3*2ndAtt+0,4*F**		**100**

Part 1

Software Management Renaissance

Introduction

- In the past ten years, typical goals in the software process improvement of several companies are to achieve a 2x, 3x, or 10x increase in productivity, quality, time to market, or some combination of all three, where x corresponds to how well the company does now.
- The funny thing is that many of these organizations have no idea what x is, in objective terms.

Part 1

Software Management Renaissance

Table of Contents (1)

Part 1
Software Management Renaissance
Table of Contents (2)

CHAPTER 1

Conventional Software Management

Key Points

▲ Conventional software management practices are mostly 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 Old Way

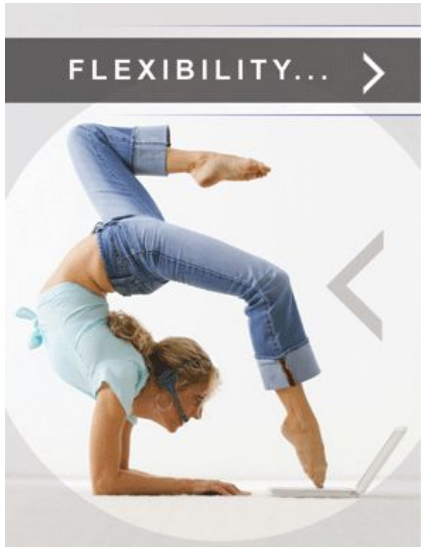
Part 1

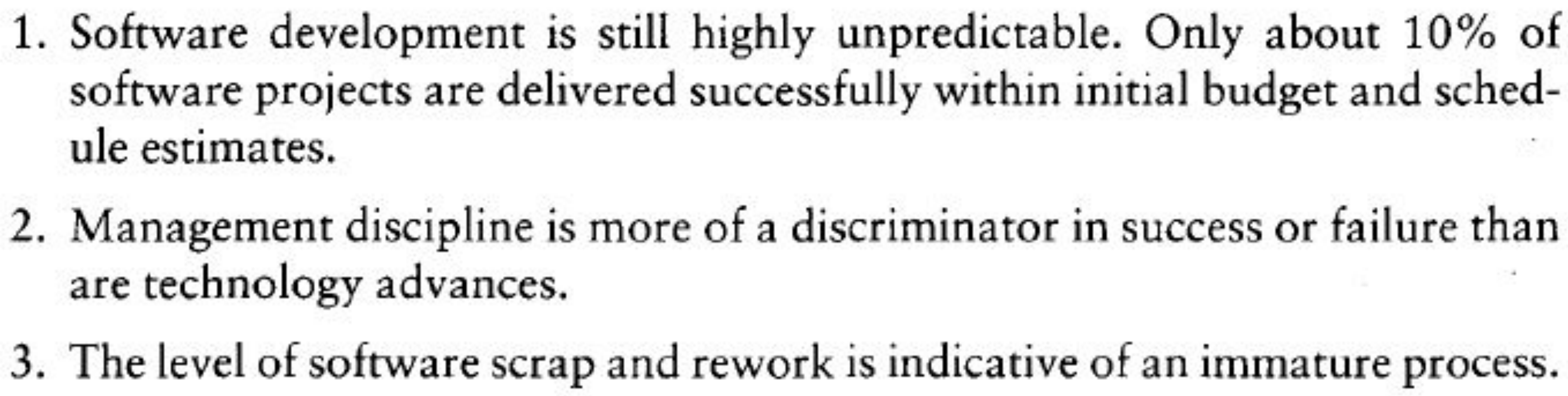
The Old Way

- Software crisis

- “The best thing about software is its flexibility”
  - It can be programmed to do almost anything.

- “The worst thing about software is also its flexibility”
  - The “almost anything ” characteristic has made it difficult to plan, monitor, and control software development.



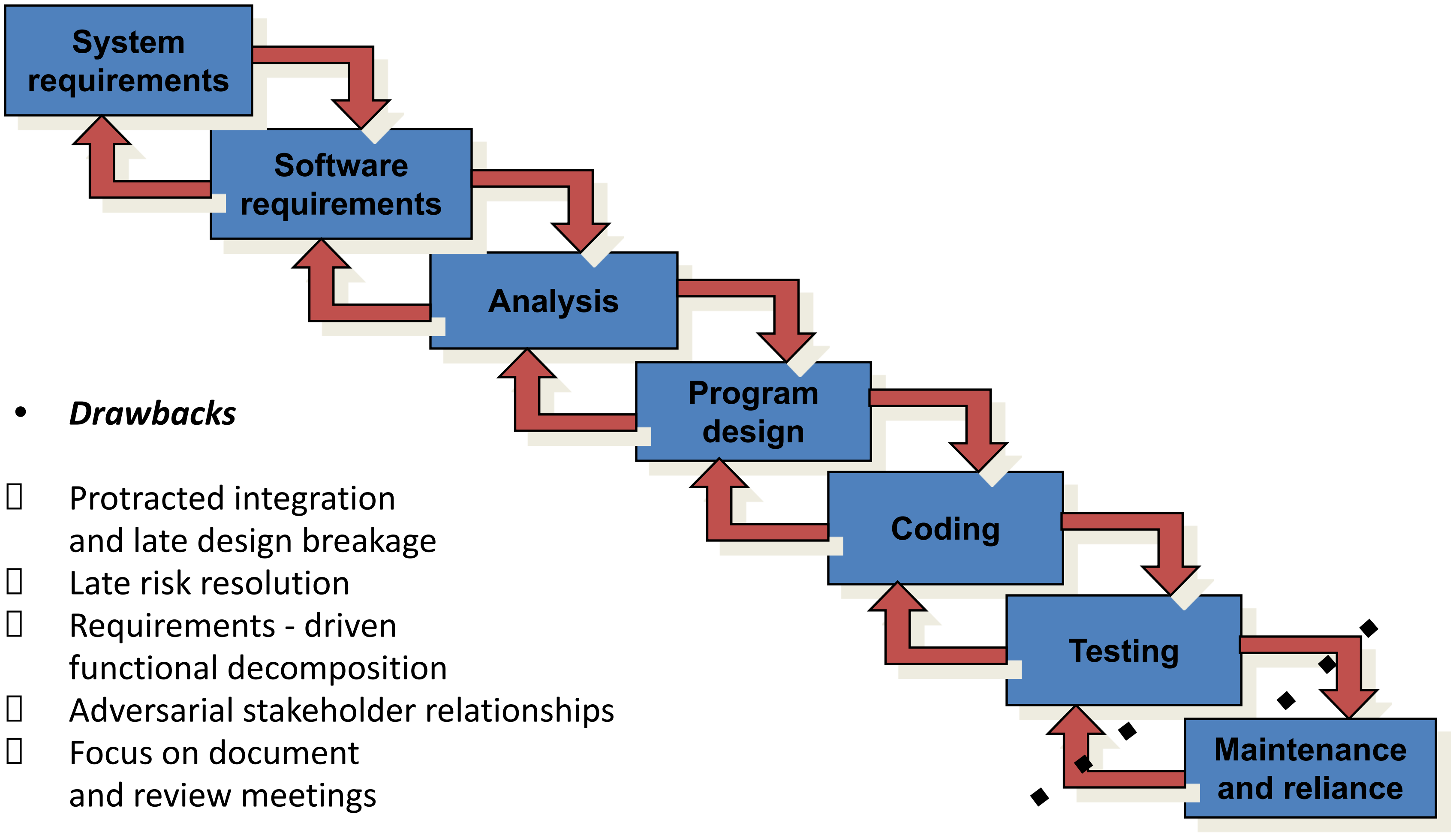


1. Software development is still highly unpredictable. Only about 10% of software projects are delivered successfully within initial budget and schedule estimates.
2. Management discipline is more of a discriminator in success or failure than are technology advances.
3. The level of software scrap and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

Part 1

The Old Way

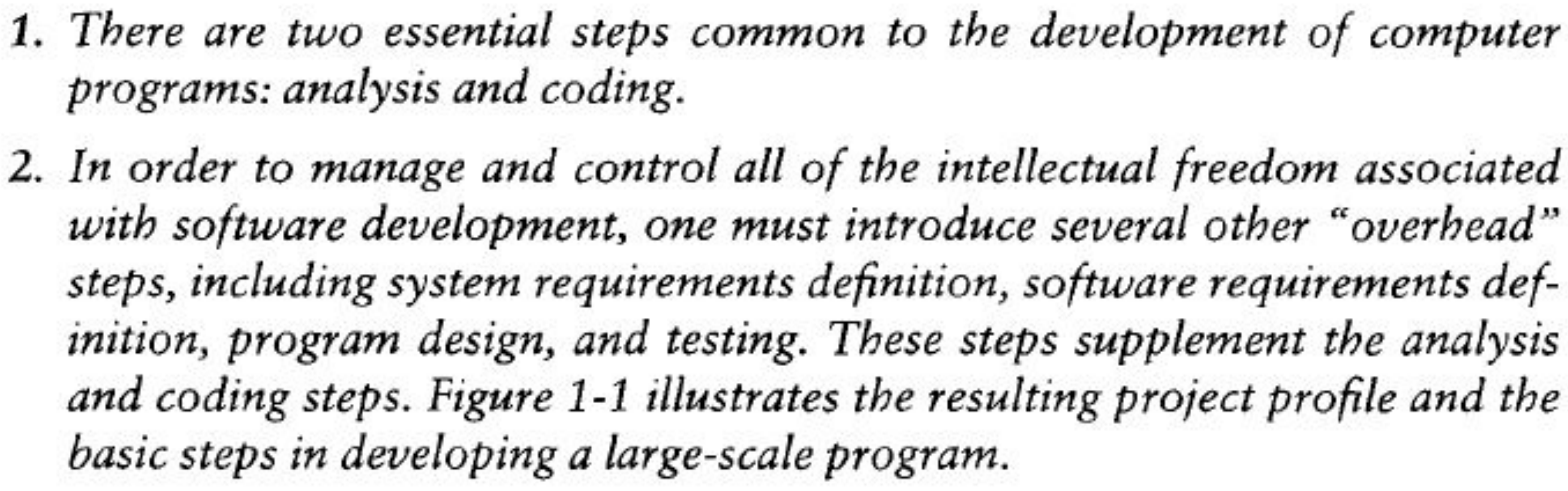
The Waterfall Model



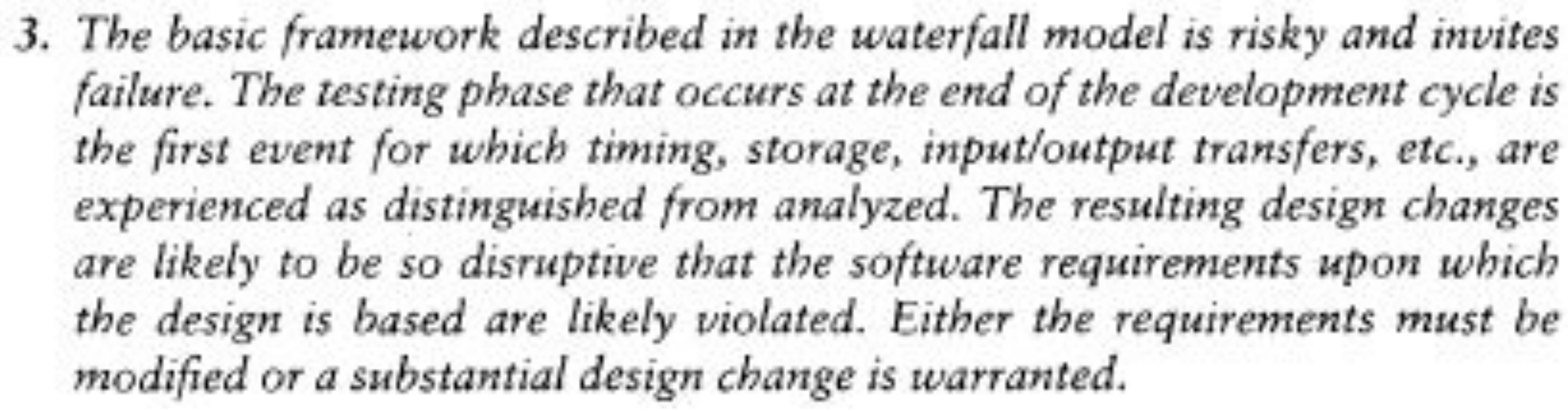
- ***Drawbacks***

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

The Waterfall Model



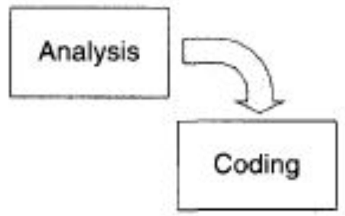
1. *There are two essential steps common to the development of computer programs: analysis and coding.*
2. *In order to manage and control all of the intellectual freedom associated with software development, one must introduce several other "overhead" steps, including system requirements definition, software requirements definition, program design, and testing. These steps supplement the analysis and coding steps. Figure 1-1 illustrates the resulting project profile and the basic steps in developing a large-scale program.*



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 from analyzed. The resulting design changes are likely to be so disruptive that the software requirements upon which the design is based are likely violated. Either the requirements must be modified or a substantial design change is warranted.*

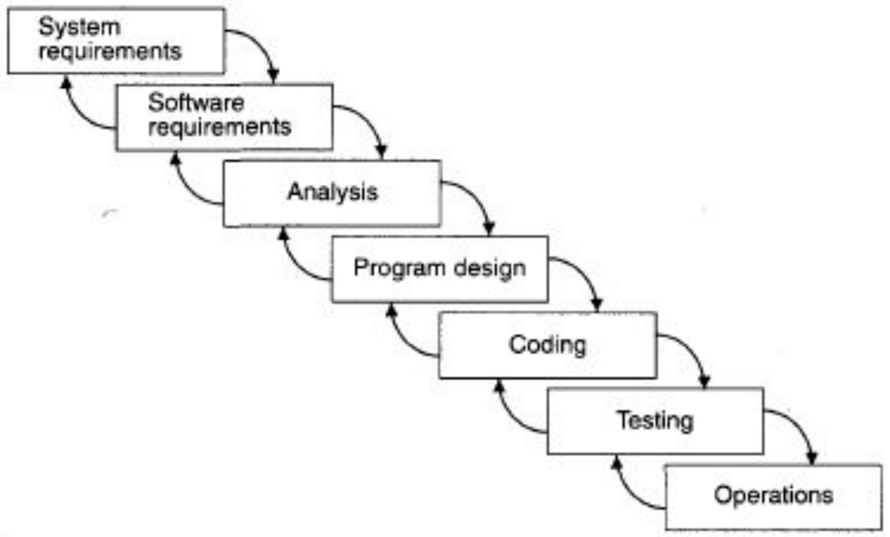
The Waterfall Model

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

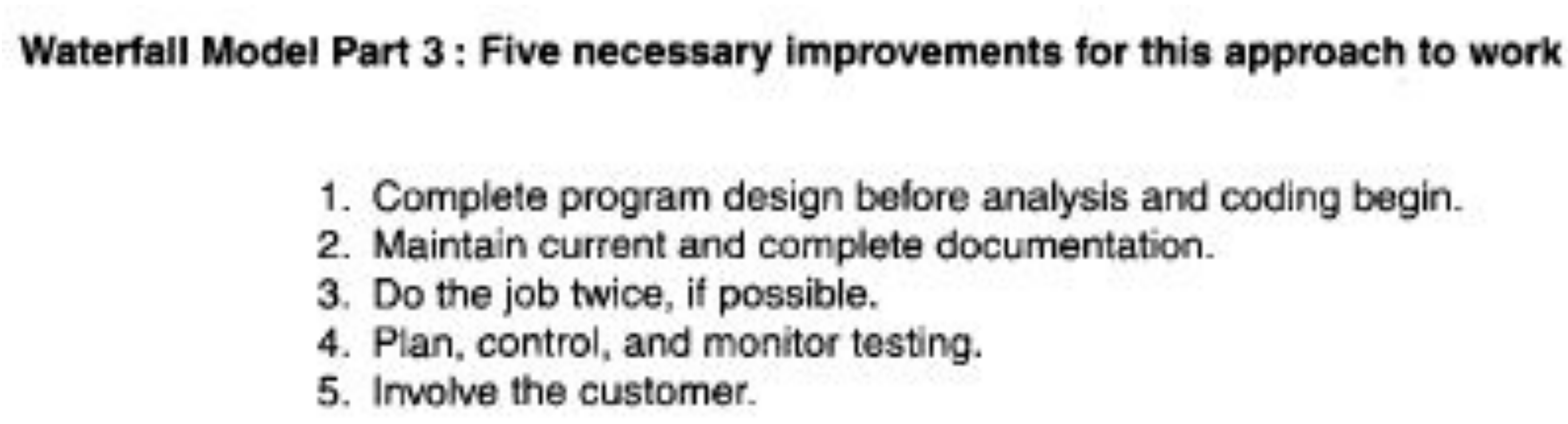


Analysis and coding both involve creative work that directly contributes to the usefulness of the end product.

Waterfall Model Part 2 : The large-scale system approach



The Waterfall Model



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.

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:

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

Part 1

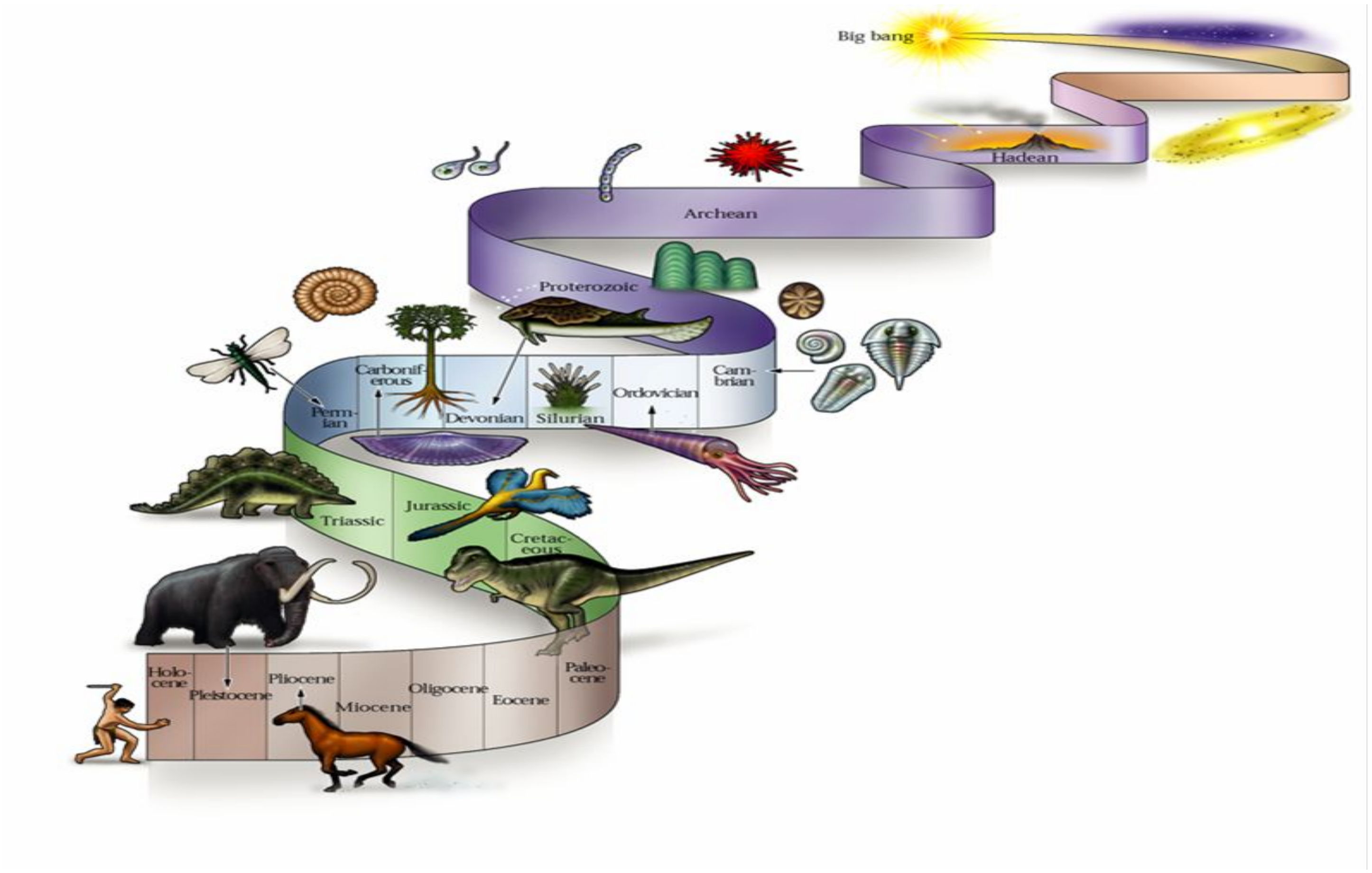
The Old Way

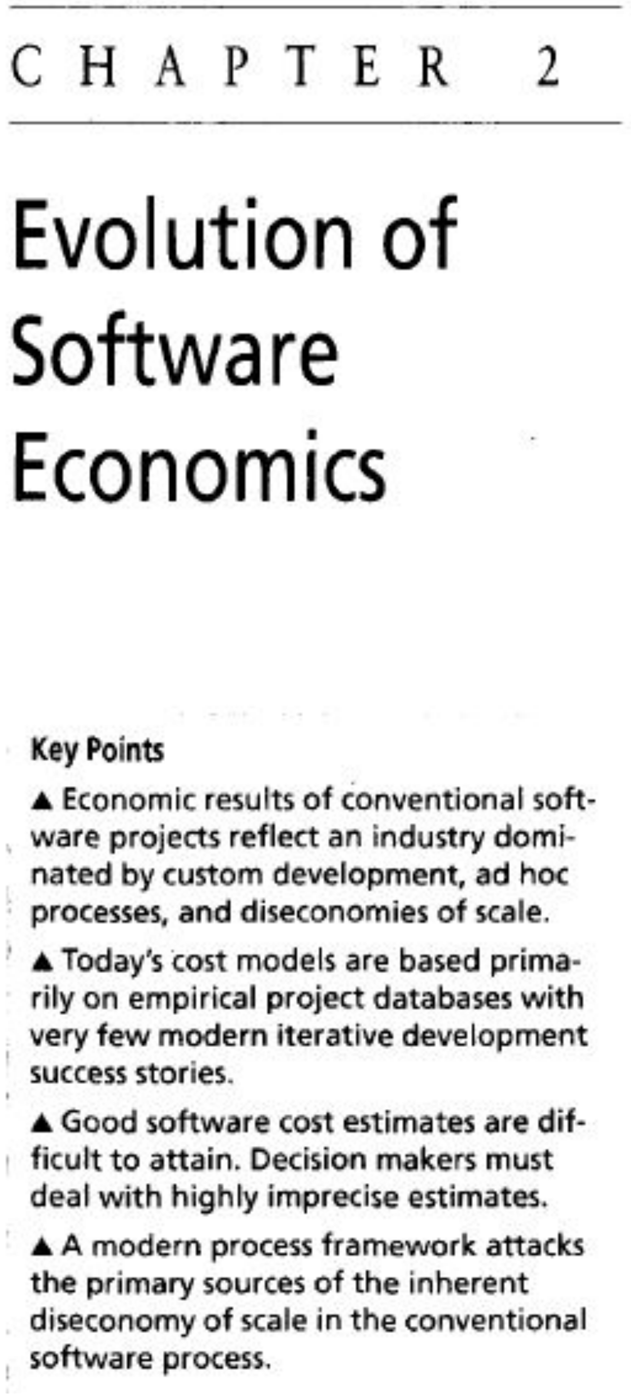
Conventional Software Management Performance

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, but no more.
3. For every \$1 you spend on development, you will spend \$2 on maintenance.
4. Software development and maintenance costs are primarily a function of the number of source lines of code.
5. Variations among people account for the biggest differences in software productivity.
6. 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. Walkthroughs catch 60% of the errors.
9. 80% of the contribution comes from 20% of contributors.

Part 1

Evolution of Software Economics





CHAPTER 2

Evolution of Software Economics

Key Points

▲ Economic results of conventional software projects reflect an industry dominated by custom development, ad hoc processes, and diseconomies of scale.

▲ Today's cost models are based primarily on empirical project databases with very few modern iterative development success stories.

▲ Good software cost estimates are difficult to attain. Decision makers must deal with highly imprecise estimates.

▲ A modern process framework attacks the primary sources of the inherent diseconomy of scale in the conventional software process.

Part 1

Evolution of Software Economics

- Most software cost models can be abstracted into a function of five basic parameters:
  - Size (typically, number of source instructions)
  - Process (the ability of the process to avoid non-value-adding activities)
  - Personnel (their experience with the computer science issues and the applications domain issues of the project)
  - Environment (tools and techniques available to support efficient software development and to automate process)
  - Quality (performance, reliability, adaptability…)

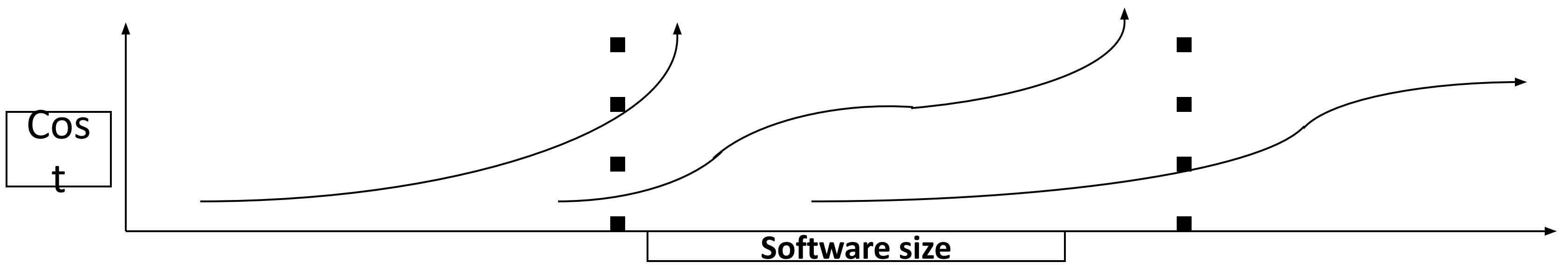
The relationships among these parameters and the estimated cost can be written as follows:

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

Part 1

Evolution of Software Economics

Three generations of software economics



1960s-1970s	1980s-1990s	2000 and on
Waterfall model Functional design Diseconomy of scale	Process improvement Encapsulation-based Diseconomy of scale	Iterative development Component- based Return to investment
Environments/tools: Custom	**Environments/tools:** Off-the-shelf, separate	**Environments/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

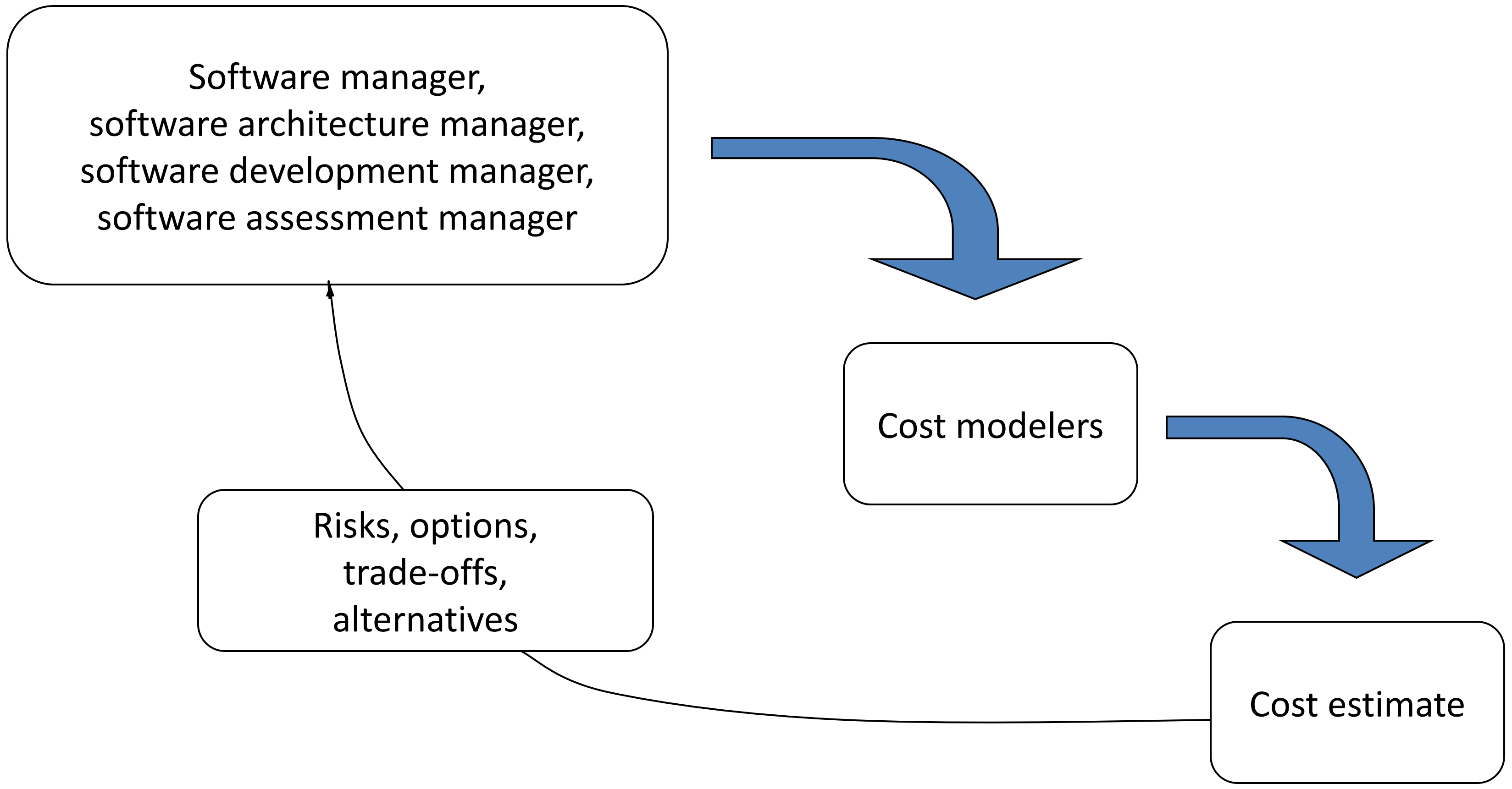
Typical project performance

Predictably bad Always: -Over budget -Over schedule	**Unpredictable** Infrequently: -On budget -On schedule	**Predictable** Usually: -On budget -On schedule

Part 1

Evolution of Software Economics

The predominant cost estimation process



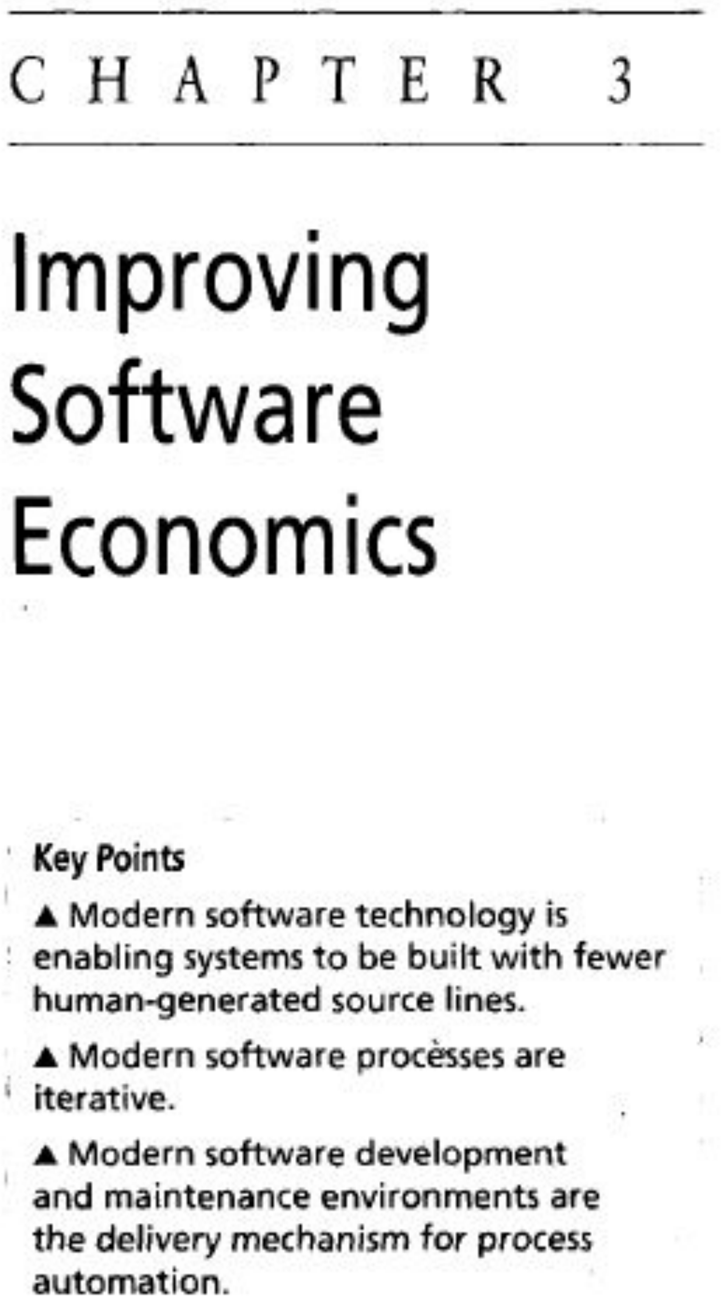
Software manager, software architecture manager, software development manager, software assessment manager
Cost modelers
Cost estimate
Risks, options, trade-offs, alternatives

Part 1

Evolution of Software Economics

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.



CHAPTER 3

Improving Software Economics

Key Points

▲ Modern software technology is enabling systems to be built with fewer human-generated source lines.

▲ Modern software processes are iterative.

▲ Modern software development and maintenance environments are the delivery mechanism for process automation.

Part 1
Improving Software Economics

- 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

Part 1

Improving Software Economics

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, analyses**
Quality Performance, reliability, accuracy	**Hardware platform performance** **Demonstration-based assessment** **Statistical quality control**

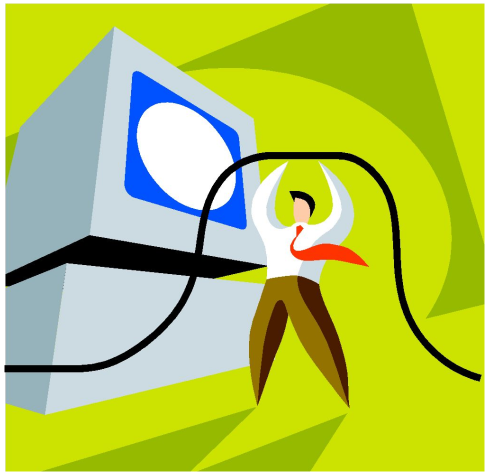
Part 1

Improving Software Economics

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.



Part 1

Improving Software Economics

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
are useful estimators for software
after a candidate solution is formulated
and
an implementation language is known.

Part 1

Improving Software Economics

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."*

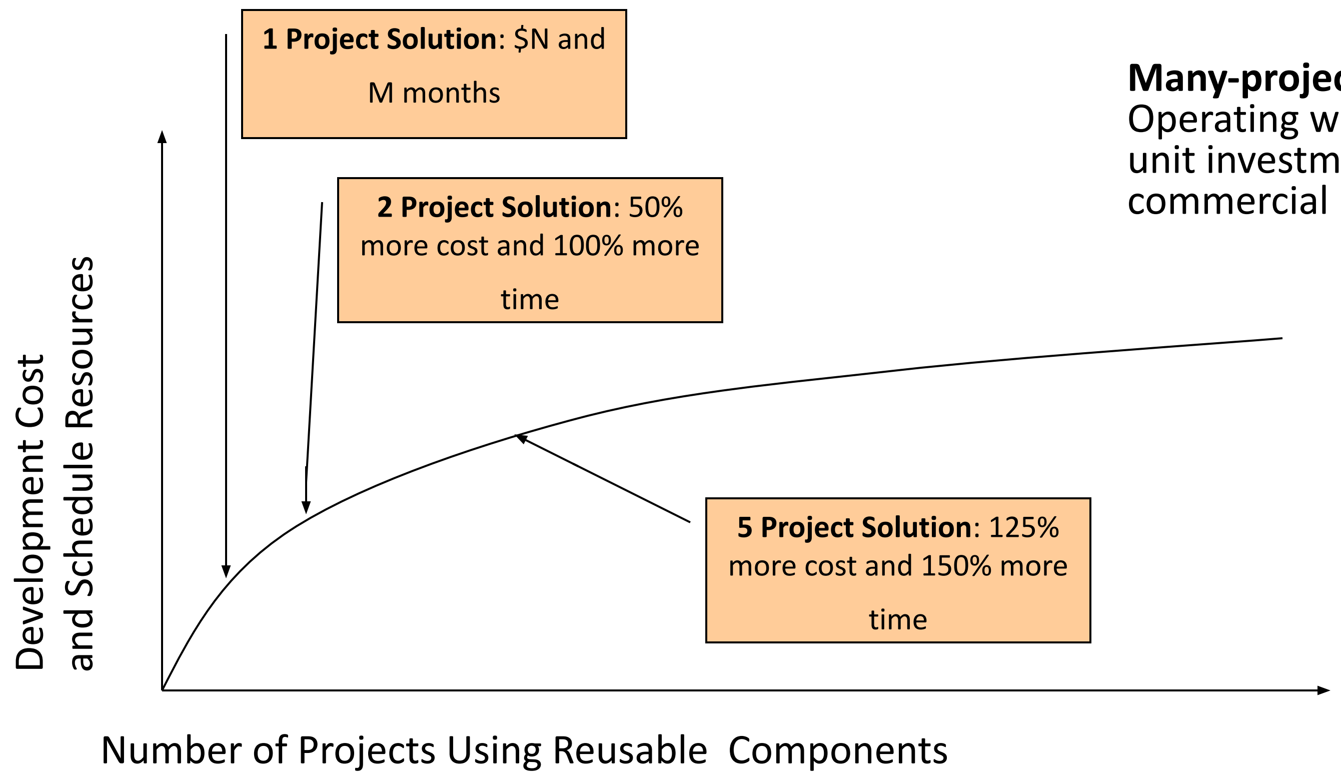
  This aspect of object-oriented technology enables an architecture-first process, in which integration is an early and continuous life-cycle activity.
- *An object-oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change 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.

Part 1

Improving Software Economics

Reducing Software Product Size – Reuse



Many-project solution: Operating with high value per unit investment, typical of commercial products

Part 1

Improving Software Economics

Reducing Software Product Size – Commercial Components

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

3.2 IMPROVING SOFTWARE PROCESSES

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

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

Part 1

Improving Software Economics

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

PRINCIPLES

It's Not The Principle That Keeps You Going
It's The Money That They Pay You

Part 1

Improving Software Economics

Improving Team Effectiveness (1)

- 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.*

Part 1

Improving Software Economics

Improving Team Effectiveness (2)

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.

Part 1

Improving Software Economics

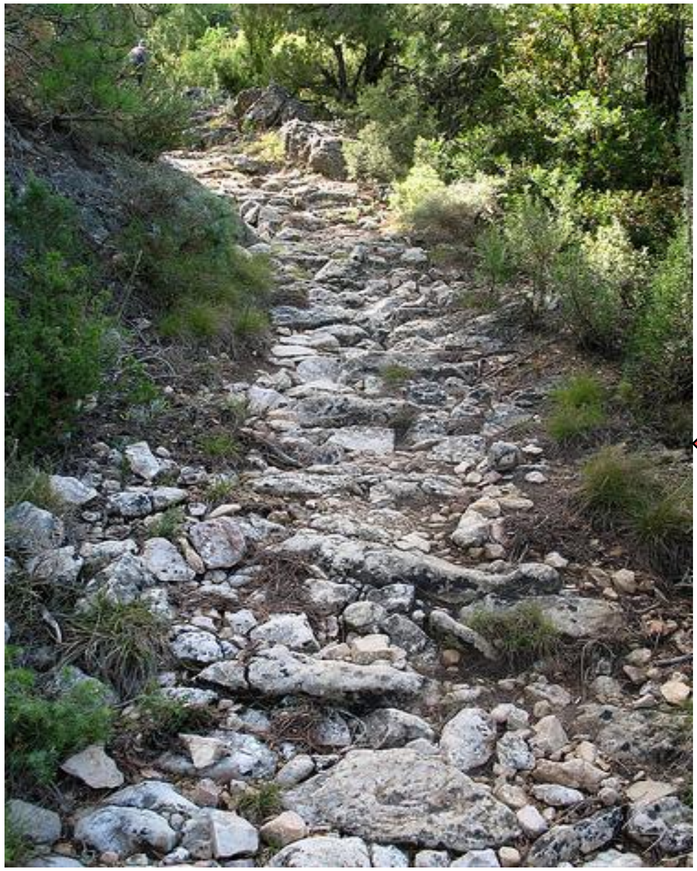
Achieving Required Quality

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

Part 1

The Old Way and the New



Part 1

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.
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.

Part 1

The Old Way and the New

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.
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."

Part 1

The Old Way and the New

The Principles of Conventional Software Engineering

21. **Use tools, but be realistic**. Software tools make their users more efficient.
22. **Avoid tricks**. Many programmers love to create programs with tricks- constructs that perform a function correctly, but in an obscure way. Show the world how smart you are by avoiding tricky code.
23. **Encapsulate**. Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.
24. **Use coupling and cohesion**. Coupling and cohesion are the best ways to measure software's inherent maintainability and adaptability.
25. **Use the McCabe complexity measure**. Although there are many metrics available to report the inherent complexity of software, none is as intuitive and easy to use as Tom McCabe's.
26. **Don't test your own software**. Software developers should never be the primary testers of their own software.
27. **Analyze causes for errors**. It is far more cost-effective to reduce the effect of an error by preventing it than it is to find and fix it. One way to do this is to analyze the causes of errors as they are detected.
28. **Realize that software's entropy increases**. Any software system that undergoes continuous change will grow in complexity and 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.

Part 1

The Old Way and the New

The Principles of Modern Software Management

Architecture-first approach → The central design element

Design and integration first, then production and test

Iterative life-cycle process → The risk management element

Risk control through ever-increasing function, performance, quality

Component-based development → The technology element

Object-oriented methods, rigorous notations, visual modeling

Change management environment → The control element

Metrics, trends, process instrumentation

Round-trip engineering → The automation element

Complementary tools, integrated environments

MCQ 1

1. Software development is unpredictable because

(a) managers are very unpredictable
(b) software by its nature is highly flexible
(c) users are usually not fully cognizant of their needs
(d) programmers are very unpredictable

2. Which of the following statements are true of the 80/20 rule?

I. "Badly behaved" modules usually make up about 20 percent of the total code but make up 80 percent of the scrap and rework cost.
II. 20 percent of the people accomplish 80 percent of the progress.
III. 20 percent of requirements account for 80 percent of engineering effort.

(a) I, II, and III
(b) III only
(c) II and III only
(d) I only

3. The waterfall model

(a) surfaces risk early
(b) allows you to correct early errors with insights gained later on
(c) discourages functional decomposition
(d) focuses on documents and review meetings

4. Which of the following statements are true of conventional software project management performance?

I. Fixing software problems after delivery of the product is relatively inexpensive.
II. Variations among people account for the biggest differences in programmer productivity.
III. It worked best if 50 percent of the development effort was devoted to programming.

(a) II only
(b) I, II, and III
(c) I only
(d) I and II only

5. The success rate for software projects is very low because

(a) software development is often a tedious and time-consuming endeavor
(b) software development relies on antiquated processes
(c) project management has more to do with project success than do programmers
(d) technology improvements are not used

MCQ 2

1. A 10,000-line software solution will cost less per line than a 100,000-line software solution because

   (a) technical biases are less important on a big project
   (b) the 100,000-line solution is a bad solution
   (c) more bugs will be found in the 100,000-line solution
   (d) communications overhead is less for a smaller team

2. Function points

(a) are usually inferior to subjective cost estimates
(b) eliminate language differences in cost estimation
(c) are easy for most organizations to learn
(d) are incompatible with most modern cost models

3.The five basic variables for software cost models are which of the following?

(a) Complexity, number of contractors to employees, process, CASE tools, and required quality
(b) Size, process, personnel, environment, and required quality
(c) Size, process, personnel, CASE tools, and purchased components
(d) Source lines of code, function points, methodology, personnel, and quality

4. Complexity arises from

(a) the increase in required communications as teams become larger
(b) depending on only one person
(c) real time systems
(d) novel technology being introduced into the process

5. In modern practice, the automation of process is a first class workflow and a focus of project management attention and project resources because

(a) iterative development means each iteration will be completely independent
(b) it allows areas of the life cycle to be improved that couldn't be improved otherwise
(c) all software development activities and tools are interrelated
(d) some tools have an extremely high payback

6. Software environments without round-trip engineering

(a) do not suffer much since early artifacts are rarely referred to as a project enters later stages
(b) are less expensive
(c) reap the benefit of having simpler tools
(d) have difficulty keeping artifacts synchronized as changes occur

7. An advantage of commercial components is that they

(a) are rich in functionality
(b) undergo frequent upgrades
(c) often have better performance
(d) can be purchased from any vendor

8. Reducing size is best accomplished through

(a) the use of Java or ADA
(b) object-oriented methods
(c) component-based development
(d) hardware investments

9. An organization can make substantial improvement through

(a) using more skilled personnel and better teams
(b) improving the development process
(c) balancing its attack across the five parameters or drivers of the cost model
(d) just concentrating on size or complexity

10. Hardware advances

(a) enable improvements in software technology
(b) allow use of commercially developed components
(c) eliminate the need for software quality control
(d) eliminate the need for highly skilled personnel

MCQ 3

1.Requirements creep can be addressed by

(a) demonstration-based review
(b) incremental releases
(c) component-based development
(d) early architecture performance feedback

Correct answer is (a)
Feedback: See section 4.2, page 66 in the textbook.

2. Intermediate releases in groups of usage scenarios

(a) are only required for baselines
(b) have nothing to do with use cases
(c) eliminate the need for use cases
(d) demonstrate an evolving understanding of system requirements

3. Cost and schedule are impacted negatively by

(a) adversarial stakeholders
(b) early breakage and scrap/rework
(c) fixed requirements
(d) inadequate function

4. A demonstration-based approach

(a) makes architectural defects inevitable
(b) allows for early elimination of architectural defects
(c) requires architectural defects to be tolerated for early releases
(d) eliminates the need for a beta test

5. Change-management environments

(a) are only important for baselines
(b) require objectively controlled baselines
(c) are too expensive for small projects
(d) rely on guidelines derived from the experience of experts

6. The architecture-first approach

(a) emerges from test results over a couple of "spirals"
(b) involves design and integration first, then production and test
(c) involves metrics, trends, and process instrumentation
(d) involves object-oriented methods, rigorous notations, and visual modeling

7. Conventional project risks

(a) only apply to the waterfall method
(b) have no impact on cost, quality, and schedule
(c) are addressed through modern software process principles
(d) no longer are of concern

8. Model-based notation

(a) has little relationship to graphical design methods
(b) is more objective than human review and inspection of ad hoc design in paper documents
(c) eliminates textual notes
(d) eliminates need for human review

9. Model-based development

(a) requires visual modeling and round-trip engineering
(b) requires object-oriented methods and rigorous notation
(c) requires complementary tools and integrated environments
(d) requires visual modeling and risk control

10. Attrition of key personnel can be addressed by

(a) a very structured environment
(b) successful early iterations and trustworthy management
(c) hiring the best candidates
(d) giving most of the responsibility to a project's average performers