SOFTWARE PROJECT MANAGEMENT

Prerequisite

- Students need to have knowledge of Software Engineering and Object Oriented Analysis and Design.

Course Objectives

- To study how to plan and manage projects at each stage of the software development life cycle (SDLC).
- To train software project managers and other individuals involved in software project planning and tracking and oversight in the implementation of the software project management process.
- To understand successful software projects that support organization’s strategic goals.

Course Outcomes

- **Upon successful completion of the course, the students will be able to:**
- **CO1** - Understand the basics of software organization as related to project and process management.
- **CO2** - Recognize the basic capabilities of software project.
- **CO3** - Procure the basic steps of project planning and project management.
- **CO4** - Compare and differentiate organization structures and project structures
- **CO5** - Employ the responsibilities for tracking the software projects.
- **CO6** - Track the process automation and project control.

UNIT-I

- **Conventional Software Management:** The Waterfall Model, Conventional Software Management Performance.
- **Evolution of Software Economics:** Software Economics, Pragmatic Software Cost Estimation.

UNIT-II

- **Improving Software Economics:** Reducing Software Product Size, Improving Software Processes, Improving Team Effectiveness, Improving Automation through Software Environments, Achieving Required Quality, Peer Inspections.
- **The Old Way and the New:** The Principles of Conventional Software Engineering, The Principles of Modern Software Management, Transitioning to an Iterative Process.

UNIT-III

- **Life-Cycle Phases:** Engineering and Production Stages, Inception Phase, Elaboration Phase, Construction Phase, Transition phase.
- **Artifacts of the Process:** The Artifact Sets, Management Artifacts, Engineering Artifacts, Pragmatic Artifacts.

UNIT-IV

- **Model-Based Software Architectures**: A Management Perspective and A Technical Perspective.
- **Workflows of the Process:** Software Process Workflows, Iteration Workflows.
- **Checkpoints of the Process:** Major Milestones, Minor Milestones, Periodic Status Assessments.

UNIT-V

- **Iterative Process Planning:** Work Breakdown Structures, Planning Guidelines, The Cost and Schedule Estimating Process, the Iteration Planning Process, Pragmatic Planning.
- **Project Organizations and Responsibilities:** Line-of-Business Organizations, Project Organizations, Evolution of Organizations.

UNIT-VI

- **Process Automation:** Tools - Automation Building Blocks, The Project Environment.
- **Project Control and Process Instrumentation:** The Seven Core Metrics, Management Indicators, Quality Indicators, Life-Cycle Expectations.

TEXT BOOKS

1. Software Project Management : A Unified Framework by Walker Royce: Pearson Education, 2005.

REFERENCE BOOKS

1. Software Project Management by Bob Hughes and Mike Cotterell: Tata McGraw-Hill Edition.
2. Software Project Management by Joel Henry, Pearson Education.
3. Software Project Management in practice by Pankaj Jalote, Pearson Education, 2005.

UNIT-I

- **Conventional Software Management:** The Waterfall Model, Conventional Software Management Performance.
- **Evolution of Software Economics:** Software Economics, Pragmatic Software Cost Estimation.

UNIT-I
PART-A

Conventional Software Management: The Waterfall Model, Conventional Software Management Performance.

Conventional Software Management

- The best thing about software is its flexibility:
  - It can be programmed to do almost anything.
- The worst thing about software is its flexibility:
- The “almost anything” characteristic has made it difficult to plan, monitor, and control software development.
- This unpredictability is the basis of what has been referred to for the past 30 years as the “software crisis.”

Conventional Software Management

- In the mid-1990s, at least three important analyses of the state of the software engineering industry were performed.
- The results were presented in *Patterns of Software Systems Failure and Success* [Jones,1996], in “Chaos” [Standish Group, 1995], and in *Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially* [Defense Science Board, 1994].

Conventional Software Management

- All three analyses reached the same general conclusion:
  - The success rate for software projects is very low.
- Although the analysis had some different perspectives, their primary messages were complementary and consistent.
- 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 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.

Conventional Software Management

- The three analyses provide a good introduction to the magnitude of the software problem and the current norms for conventional software management performance.
- There is much room for improvement.

Waterfall Model

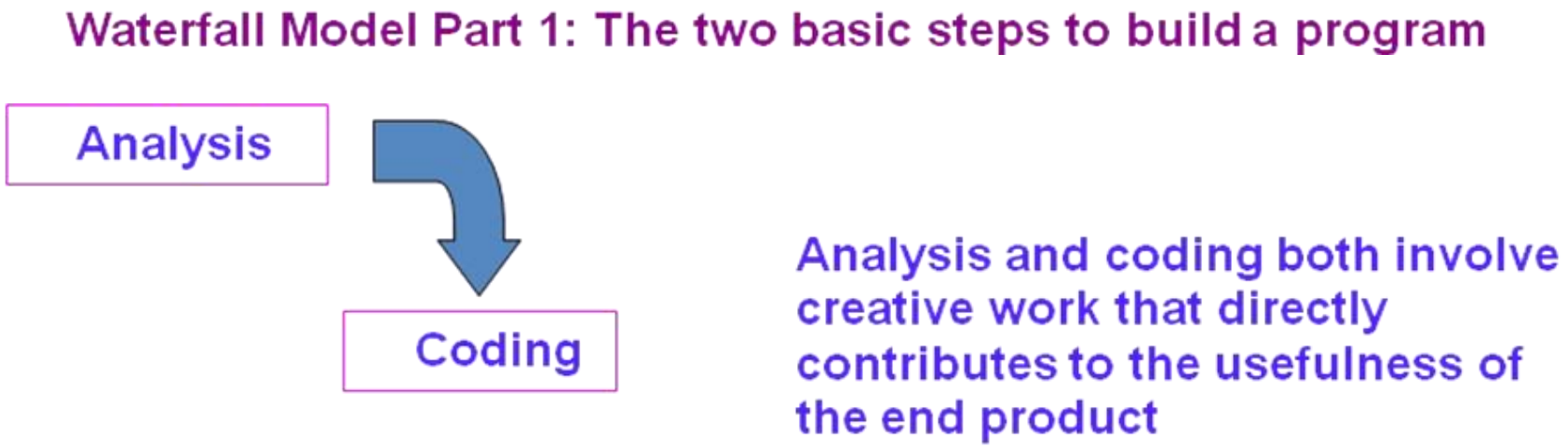
- It is the baseline process for most conventional software projects have used.
- We can examine this model in two ways:
  - IN THEORY
  - IN PRACTICE

In Theory

- In the year of 1968, Winston Royce, presented a paper titled “Managing the Development of Large Scale Software Systems” at IEEE WESCON.
- Based on this paper work his son walker Royce proposed a model “Linear Sequential Model” which is also called “waterfall model” in the year 1970.
- This model provides a road map for doing a software engineering work.
- The paper made three primary points:

In Theory

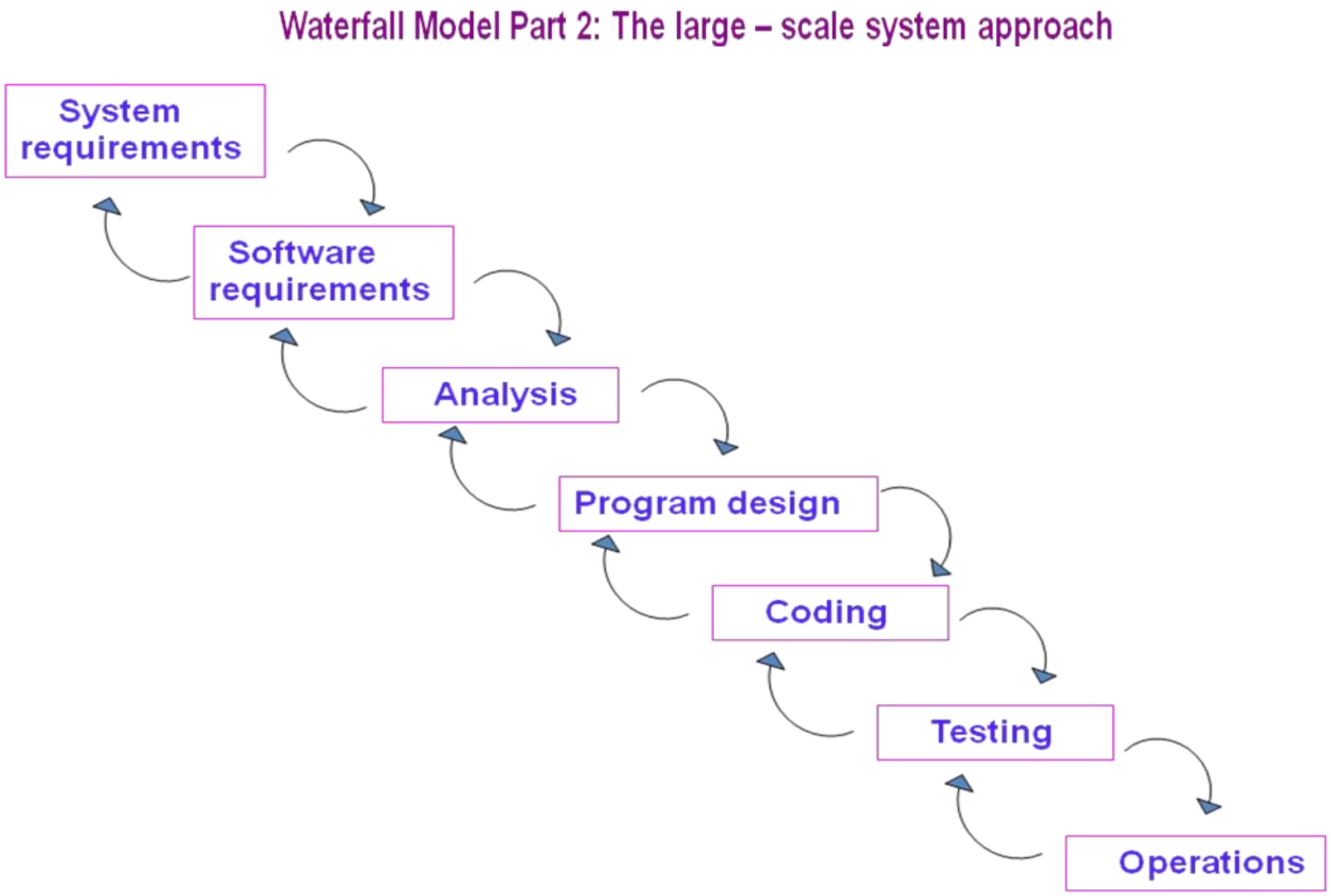
1. There are two essential steps common to the development of computer programs: analysis and coding.



In Theory

2. In order to manage and control all of the intellectual freedom associated with software development, one should follow the following steps, including System requirements definition, Software requirements definition Program design, and testing.
   - These steps supplement the analysis and coding steps.

In Theory


Waterfall Model Part 2: The large – scale system approach
System requirements
Software requirements
Analysis
Program design
Coding
Testing
Operations

In Theory

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.

In Theory

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.

In Theory

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

1. **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.

In Theory

2. **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.

In Theory

3. **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.

In Theory

4. **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.

In Theory

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

Protracted Integration and Late Design Breakage

- For a typical development project that used a waterfall model management process, below figure illustrates development progress versus time.
- Progress is defined as percent coded, i.e, demonstrable in its target form.
- 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 due to unforeseen implementation issues and interface ambiguities
  - Heavy budget and schedule pressure to get the system working
  - Late shoe-horning of nonoptimal fixes, with no time for redesign
  - A very fragile, unmaintainable product delivered late

Protracted Integration and Late Design Breakage

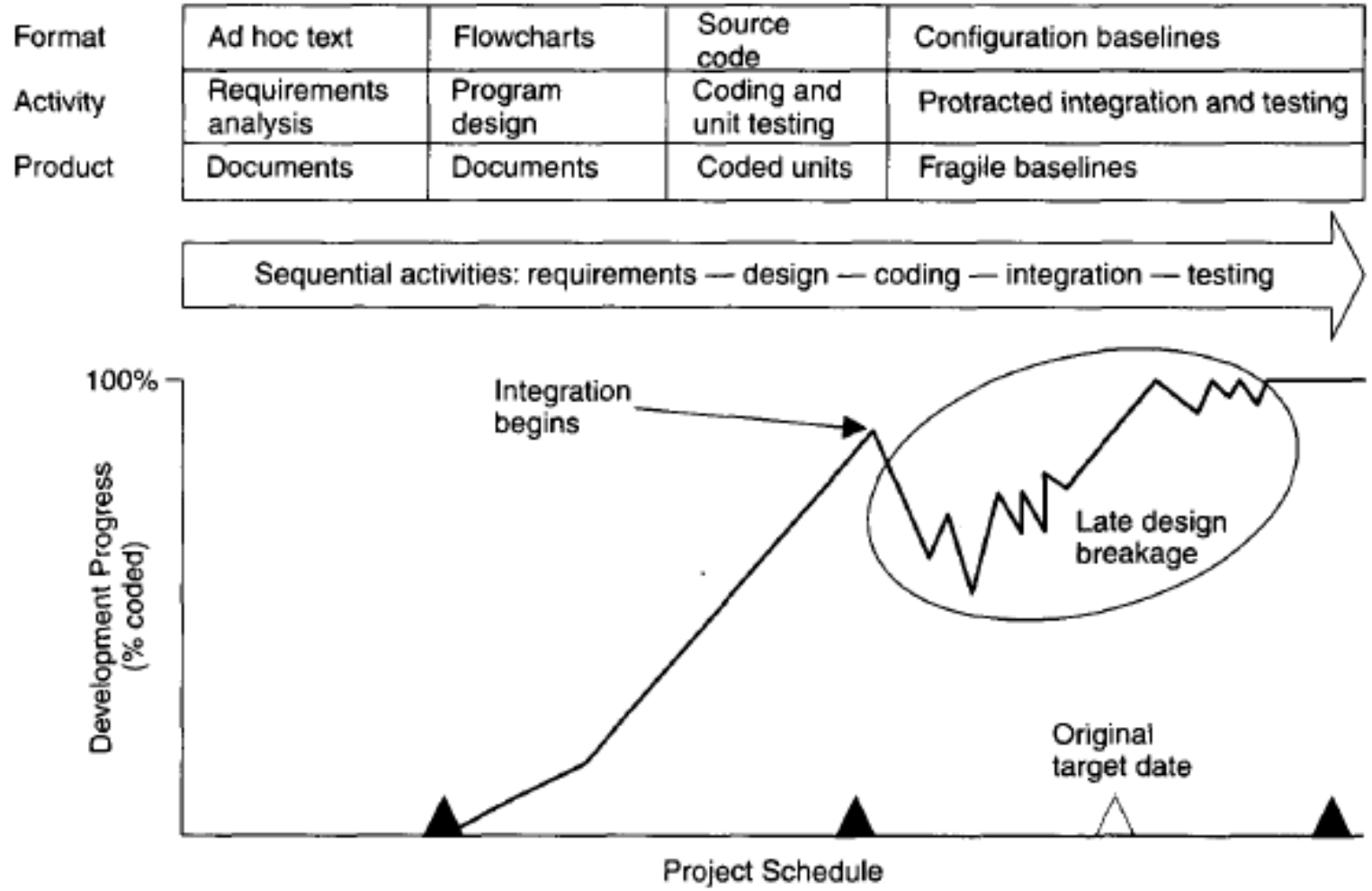


FIGURE 1-2. *Progress profile of a conventional software project*

Protracted Integration and Late Design Breakage

- Conventional techniques that imposed a waterfall model on the design process inevitably resulted in late integration and performance showstoppers.
- 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 (interfaces and structure) was sound.
- One of the recurring themes of projects following the conventional process was that testing activities consumed 40% or more of life-cycle resources.
- Below table provides a typical profile of cost expenditures across the spectrum of software activities.

Protracted Integration and Late Design Breakage

TABLE 1-1. *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%

Late Risk Resolution

- A serious issue associated with the waterfall life cycle was the lack of early risk resolution.
- This was not so much a result of the waterfall life cycle as it was of the focus on early paper artifacts, in which the real design, implementation, and integration risks were still relatively intangible.
- Below figure illustrates a typical risk profile for conventional waterfall model projects.
- It includes four distinct periods of risk exposure, where risk is defined as the probability of missing a cost, schedule, feature, or quality goal.
- Early in the life cycle, as the requirements were being specified, the actual risk exposure was highly unpredictable.

Late Risk Resolution

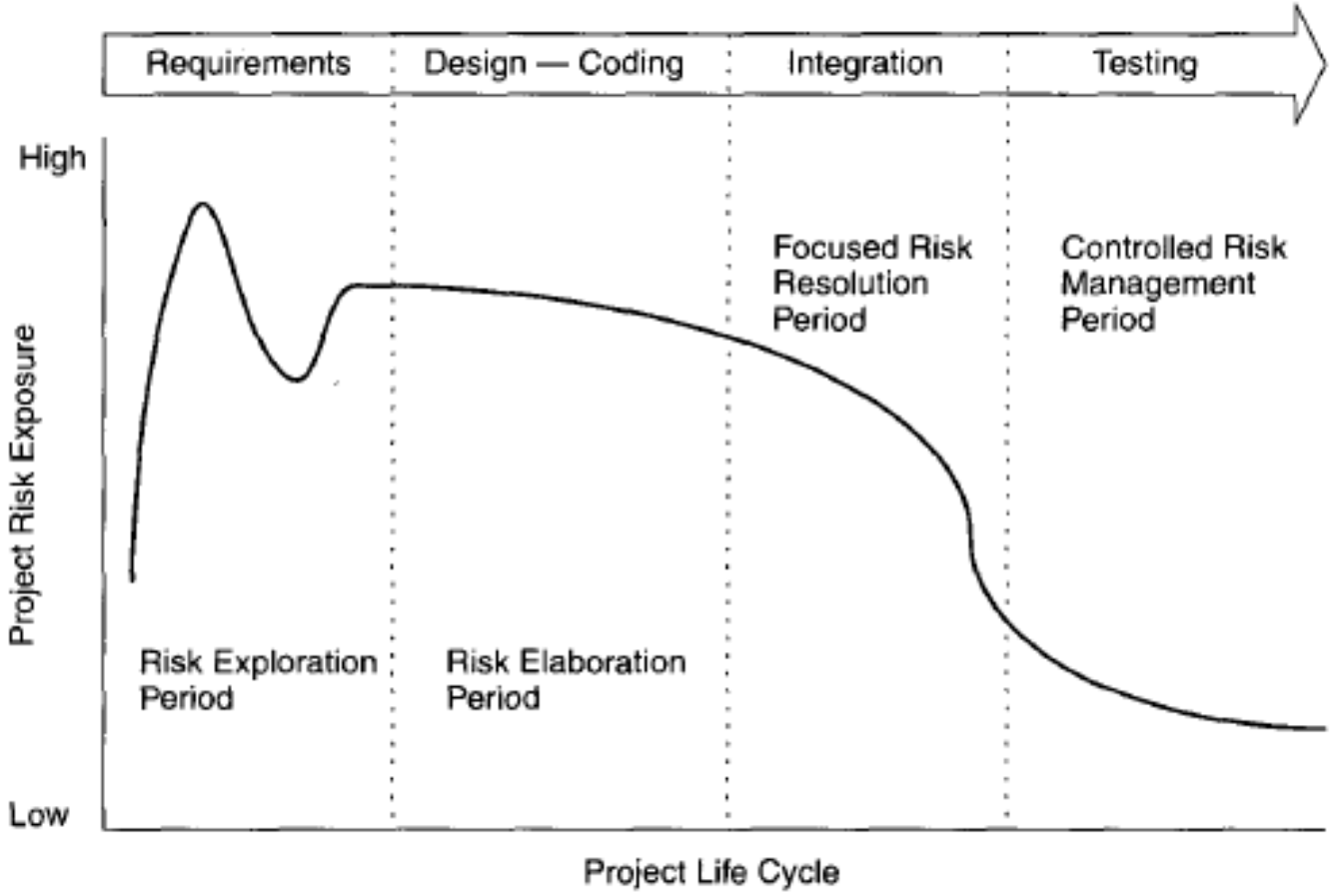


FIGURE 1-3. *Risk profile of a conventional software project across its life cycle*

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.

Requirements-Driven Functional Decomposition

- Another property of the conventional approach is that the requirements were typically specified in a functional manner.
- Built into the classic waterfall process was the fundamental assumption that the software itself was decomposed into functions; requirements were then allocated to the resulting components.
- This decomposition was often very different from a decomposition based on object-oriented design and the use of existing components.
- The functional decomposition also became anchored in contracts, subcontracts, and work breakdown structures, often precluding a more architecture-driven approach.
- Below figure illustrates the result of requirements-driven approaches: a software structure that is organized around the requirements specification structure.

Requirements-Driven Functional Decomposition

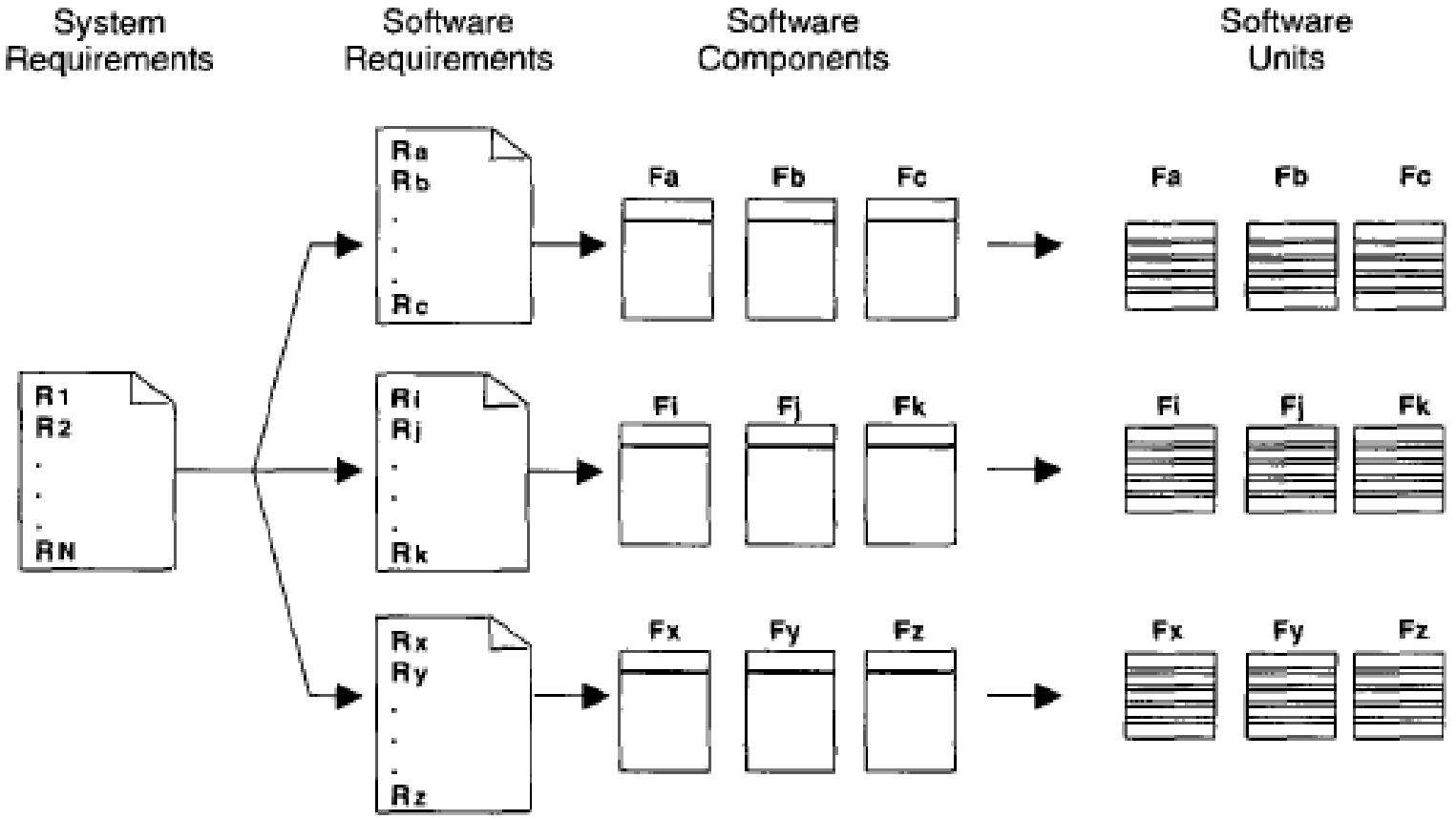


FIGURE 1-4. *Suboptimal software component organization resulting from a requirements-driven approach*

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

Focus on Documents and Review Meetings

- The conventional process focused on producing 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.
- Typically, presenters and the audience reviewed the simple things that they understood rather than the complex and important issues.
- Most design reviews resulted in low engineering and high cost in terms of the effort and schedule involved in their preparation and conduct.

Focus on Documents and Review Meetings

TABLE 1-2. *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.

Conventional Software Management Performance

- Barry Boehm's one-page "Industrial Software Metrics Top 10 List" is a good, objective characterization of the state of software development.
- Although many of the metrics are gross generalizations, they accurately describe some of the fundamental economic relationships that resulted from the conventional software process practiced over the past 30 years.

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 (small), but no more.
3. For every $1 you spend on development, you will spend $2 on maintenance.

Conventional Software Management Performance

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.

Conventional Software Management Performance

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 (i.e., system of systems) cost 9 times as much.
9. Walkthroughs catch 60% of the errors.

Conventional Software Management Performance

10. 80% of the contribution comes from 20% of the contributors.

The following fundamental postulates underlie the rationale for a modern software management process framework:

- ✓ 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.