What is software project management?

Software project management (SPM):

- SPM is an appropriate way of planning software projects.
- SPM is a part of project management where projects are planned, designed, implemented, checked, and maintained.

Need of SPM:

- Software development is a new stream in the software business, and there is very little experience in developing software.
- Most software is developed to fit the requirements of the client.
- The technology advances and changes so rapidly that the experience of one software product cannot be applied to another software product.
- Such types of constraints can raise risks in the development of software.
- It is also essential for a business in delivering quality software, keeping the product cost within the budget of the customer, and delivering the software as per schedule.

The goal of software project management is to understand, plan, measure and control the project such that it is delivered on time and on budget. This involves gathering requirements, managing risk, monitoring and controlling progress, and following a software development process.

Software project management is extremely important for the following reasons:

- **Software development is highly unpredictable: [as of 2007] only about 10% of projects are delivered within initial budget and on schedule.**
- **Management has a greater effect on the success or failure of a project than technology advances.**
- **Too often there is too much scrap and rework. The entire process is very immature, not enough reuse**
- Project failure in itself is not the only reason why software management is so important. When a project fails, not only is a product not delivered, but all the money invested in the product is also lost. Without proper software management, even completed projects will be delivered late and over budget.

- **Conventional Software:** Conventional software is typically a software application that can ccomplish some specific tasks.
- For instance, a web browser is a conventional software. **Web What is conventional software management?browser:** A web browser is a conventional software that is utilized for accessing and viewing websites; some of the typical web browsers are listed below:

- Microsoft Internet Explorer.
- Mozilla Firefox
- Google Chrome.
- Apple Safari.

  The prime function of a web browser is to deliver Hypertext Markup Language (HTML), the code utilized for designing webpages.
- Every time a web browser loads a webpage, it processes the HTML code, which includes the text, references, links, cascading style sheets (CSS), and JavaScripts.
  - For instance, Ajax facilitates a web browser to update information dynamically on a webpage without the requirement of reloading the webpage.

Conventional software management:

- In the past, organizations used conventional software management.
- This management utilized custom tools and process and virtually custom components built-in primitive languages.
  - Thus, the performance of the project was very much predictable in the schedule, cost, and quality.
- It is a practically outdated technique and technology.
  - The best thing about conventional software is the flexibility of software.
  - The worst thing about conventional software is also the flexibility of software.
- **There are three primary analyses of a software industry's state, and they are listed below:**
  - **The development of software is still unpredictable.**
  - **Management regulation is a discriminator in the failure or success.**
  - **The level of rework and software scrap are indicators of an undeveloped process.**

  - **WaterfallModel:**

    The Waterfall Model was the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

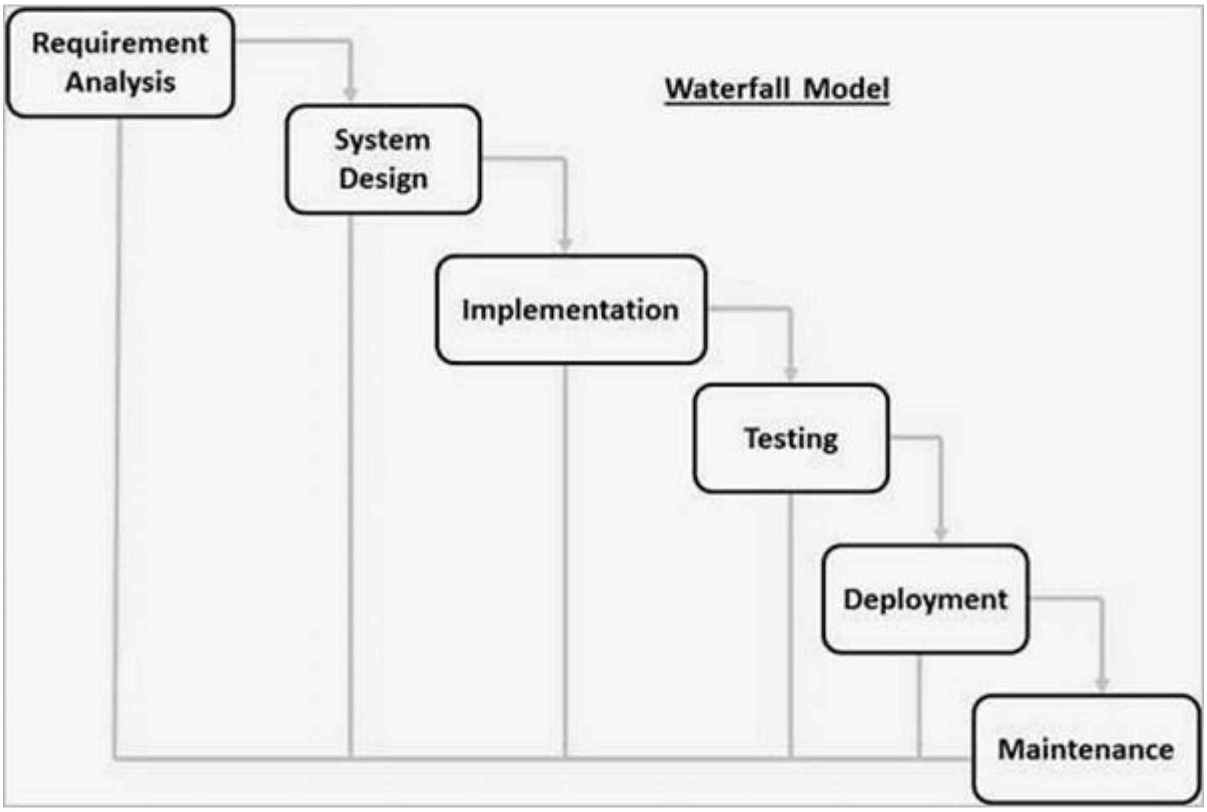
The Waterfall model is the earliest SDLC approach that was used for software development.

The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

Waterfall Model - Design

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are −

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system

design helps in specifying hardware and system requirements and helps in defining the overall system architecture.

- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model, phases do not overlap.

Waterfall Model - Application

Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are −

- Requirements are very well documented, clear and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available to support the product.
- The project is short.

Waterfall Model - Advantages

The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.

Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.

Some of the major advantages of the Waterfall Model are as follows −

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

Waterfall Model - Disadvantages

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

The major disadvantages of the Waterfall Model are as follows −

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.

Five necessary improvements for waterfall model are:-

1. **Program design comes first**: Insert a preliminary program design phase between the software requirements generation phase and the analysis phase. **By this technique, the program designer assures that the software will not fail because of storage, timing, and data flux (continuous change).** As analysis proceeds in the succeeding phase, the program designer must impose on the analyst the storage, timing, and operational constraints in such a way that he senses the consequences. The following steps are required:

   Begin the design process with program **designers**, not analysts or programmers.

   Design, define, and allocate the data processing modes even at the risk of being wrong. Allocate processing functions, design the database, allocate execution time, define interfaces and processing modes with the operating system, describe input and output processing, and define preliminary operating procedures.

   .

2.Document the design. The amount of documentation required on most software programs is quite a lot, certainly much more than most programmers, analysts, or program designers are willing to do if left to their own devices. Why do we need so much documentation? **(1)** Each designer must communicate with interfacing designers, managers, and possibly customers. **(2)** During early phases, the documentation is the design. (3) The real monetary value of documentation is to support later modifications by a separate test team, a separate maintenance team, and operations personnel who are not software literate.

3. **Do it twice.** If a computer program is being 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. Note that this is simply the entire process done in miniature, to a time scale that is relatively small with respect to the overall effort. In the first version, the team must have a special broad competence where they can quickly sense trouble spots in the design, model them, model alternatives, forget the straightforward aspects of the design that aren't worth studying at this early point, and, finally, arrive at an error-free program.

4. **Plan, control, and monitor testing**. Without question, the biggest user of project resources-manpower, computer time, and/or management judgment-is the test phase. This is the phase of greatest risk in terms of cost and

schedule. It occurs at the latest point in the schedule, when backup alternatives are least available, if at all. The previous three recommendations were all aimed at uncovering and solving problems before entering the test phase. However, even after doing these things, there is still a test phase and there are still important things to be done, including: (1) employ a team of test specialists who were not responsible for the original design; (2) employ visual inspections to spot the obvious errors like dropped minus signs, missing factors of two, jumps to wrong addresses (do not use the computer to detect this kind of thing, it is too expensive); (3) test every logic path; (4) employ the final checkout on the target computer.

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. There are three points following requirements definition where the insight, judgment, and commitment of the customer can bolster the development effort. These include a "preliminary software review" following the preliminary program design step, a sequence of "critical software design reviews" during program design, and a "final software acceptance review".

IN PRACTICE

Some software projects still practice the conventional software management approach.

It is useful to summarize the characteristics of the conventional process as it has typically been applied, which is not necessarily as it was intended. Projects destined for trouble frequently exhibit the following symptoms:

- Protracted integration and late design breakage.
- Late risk resolution.
- Requirements-driven functional decomposition.
- Adversarial (conflict or opposition) stakeholder relationships.
- Focus on documents and review meetings.

Protracted Integration and Late Design Breakage

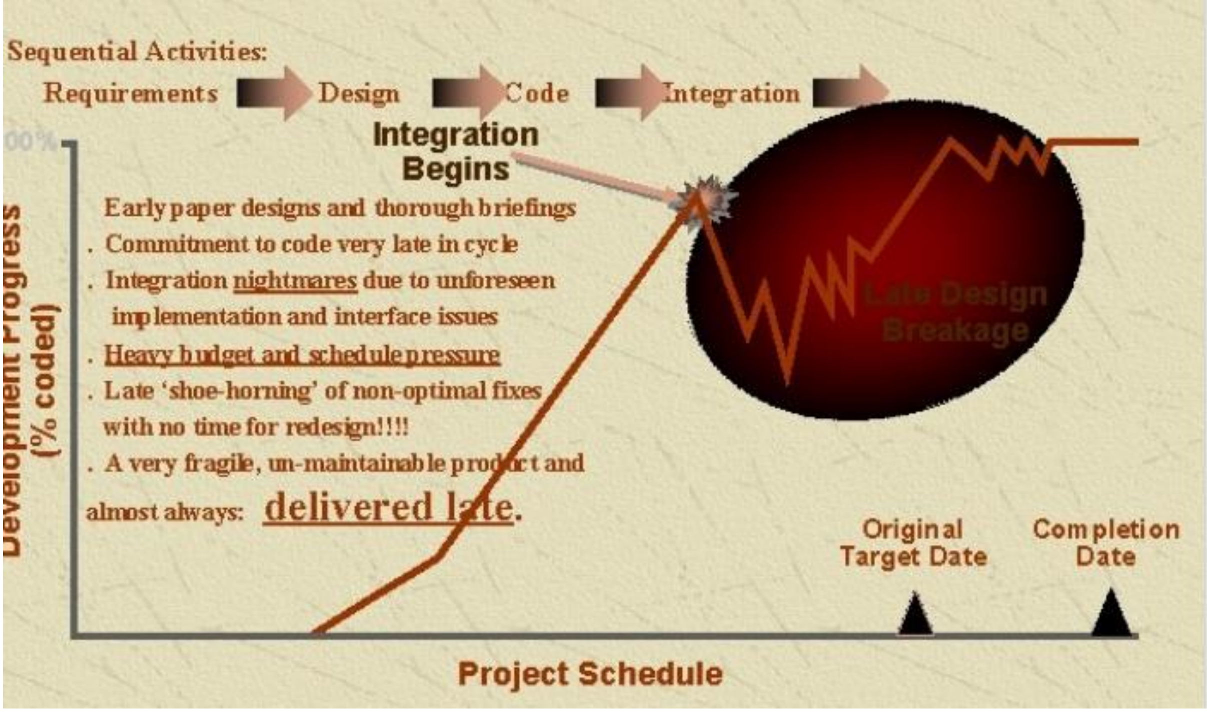
For a typical development project that used a waterfall model management process, Figure 1-2 illustrates development progress versus time. Progress is defined as percent coded, that is, demonstrable in its target form.

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 (unpleasant experience) due to unforeseen

implementation issues and interface ambiguities.

- Heavy budget and schedule pressure to get the system working.
- Late shoe-homing of no optimal fixes, with no time for redesign.
- A very fragile, unmentionable product delivered late.

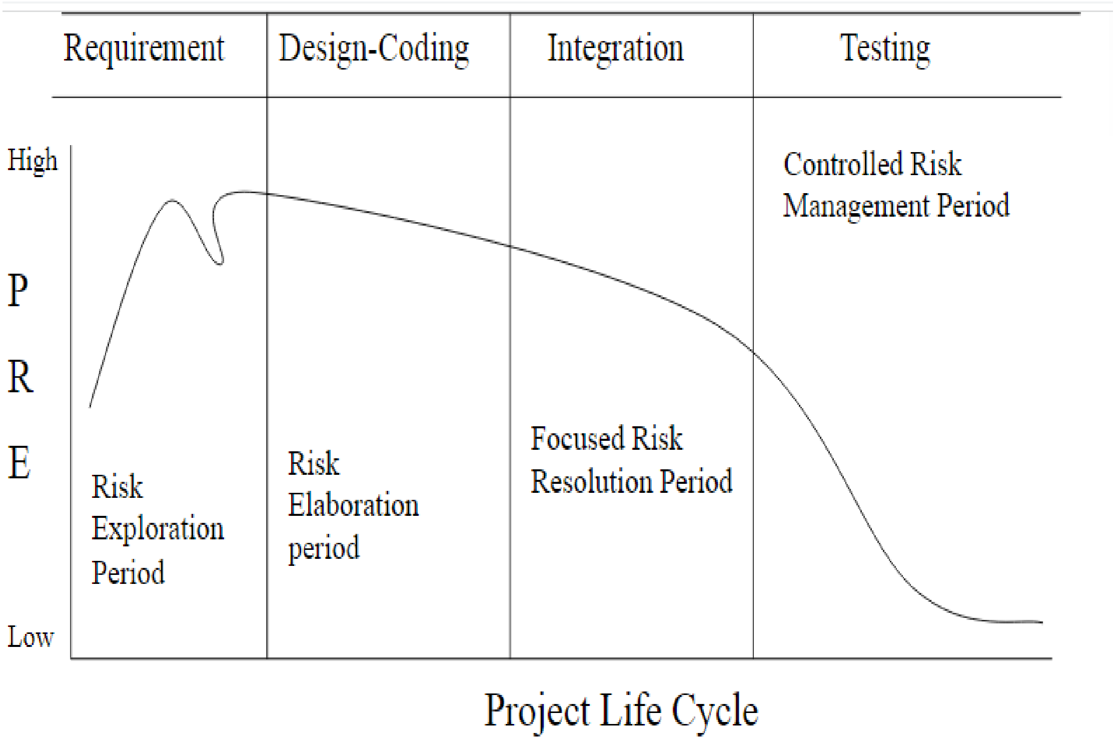


In the conventional model, the entire system was designed on paper, then implemented all at once, then integrated. Table 1-1 provides a typical profile of cost expenditures across the spectrum of software activities.

Activity	Cost
Management	5%
Requirements	5%
Design	**10%**
Code and unit test	**30%**
Integration and Test	**40%**
Deployment	**5%**
Environment	**5%**
Total	**100%**

Table1:Expenditures per activity for a Conventional Software Project

Late risk resolution A serious issue associated with the waterfall lifecycle was the lack of early risk resolution. Figure 1.3 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.



Requirements-Driven Functional Decomposition: This approach depends on specifying requirements com-pletely and unambiguously before other development activities begin. It naively treats all requirements as equally important, and depends on those requirements remaining constant over the software development life cycle. These conditions rarely occur in the real world. Specification of requirements is a difficult and important part of the software development process.

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. Figure 1-4 illustrates the result of requirements-driven approaches: a software structure that is organized around the requirements specification structure.

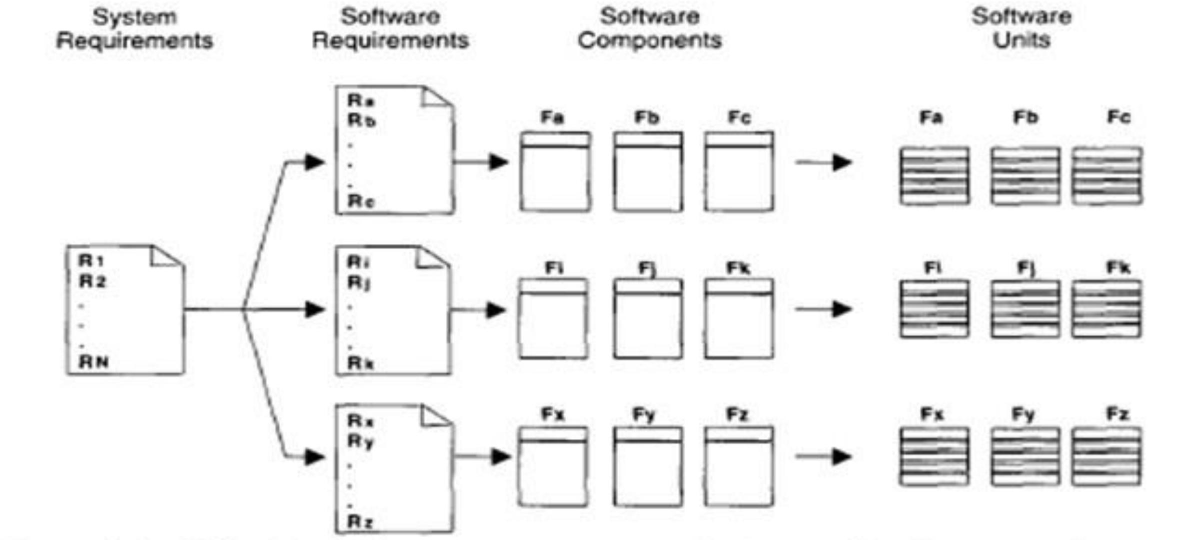


FIGURE 1-4. *Suboptimal software component organization resulting from a requirements-driven approach*

1.2 CONVENTIONAL SOFTWARE MANAGEMENT PERFORMANCE:

Barry Boehm's "Industrial Software Metrics Top 10 List” is a good, objective characterization of the state of software development.

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

2. You can compress software development schedules 25% of nominal, 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. 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

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

2.Evolution of Software Economics

2.1 SOFTWARE ECONOMICS

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

1. The ***size*** of the end product (in human-generated components), which is typically quantified in terms of the number of source instructions or the number of function points required to develop the required functionality
2. The ***process*** used to produce the end product, in particular the ability of the process to avoid non-value-adding activities (rework, bureaucratic delays, communications overhead)
3. The capabilities of software engineering ***personnel***, and particularly their experience with the computer science issues and the applications domain issues of the project
4. The ***environment***, which is made up of the tools and techniques available to support efficient software development and to automate the process
5. The required ***quality*** of the product, including its features, performance, reliability, and adaptability

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

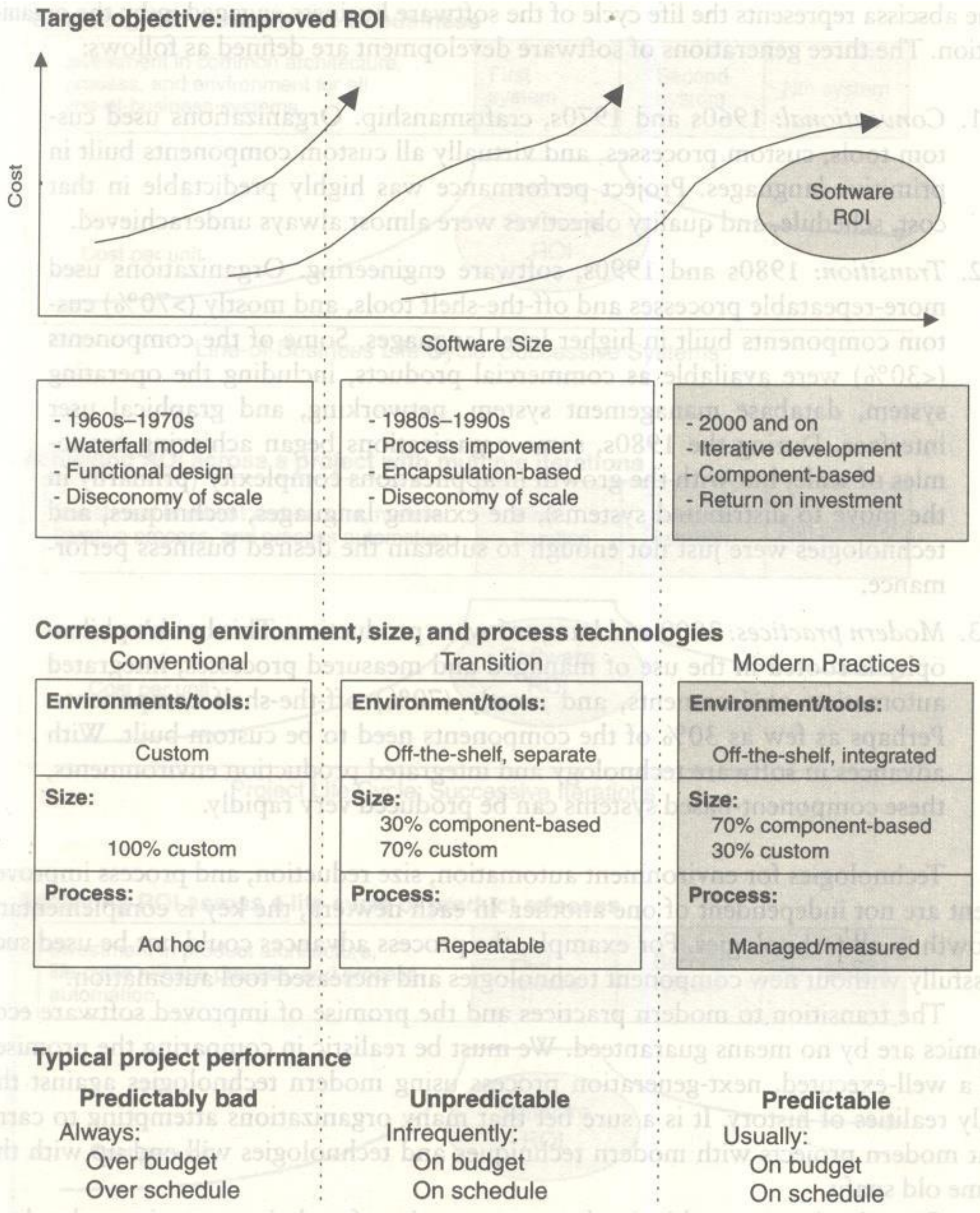
Effort = (Personnel) (Environment) (Quality) (Sizeprocess)

One important aspect of software economics (as represented within today's software cost models) is that the relationship between effort and size exhibits a diseconomy of scale. The diseconomy of scale of software development is a result of the process exponent being greater than 1.0. Contrary to most manufacturing processes, the more software you build, the more expensive it is per unit item.

Figure 2-1 shows three generations of basic technology advancement in tools, components, and processes. The required levels of quality and personnel are assumed to be constant. The ordinate of the graph refers to software unit costs (pick your favorite: per SLOC, per function point, per component) realized by an organization.

The three generations of software development are defined as follows:

1) ***Conventional:*** 1960s and 1970s, craftsmanship. Organizations used custom tools, custom processes, and virtually all custom components built in primitive languages. Project performance was highly predictable in that cost, schedule, and quality objectives were almost always underachieved.
2) ***Transition****:* 1980s and 1990s, software engineering. Organiz:1tions used more-repeatable processes and off-the-shelf tools, and mostly (>70%) custom components built in higher level languages. Some of the components (<30%) were available as commercial products, including the operating system, database management system, networking, and graphical user interface.
3) ***Modern practices****:* 2000 and later, software production. This book's philosophy is rooted in theuse of managed and measured processes, integrated automation environments, and mostly(70%) off-the-shelf components. Perhaps as few as 30% of the components need to be custombuilt Technologies for environment automation, size reduction, and process improvement are not independent of one another. In each new era, the key is complementary growth in all technologies. For example, the process advances could not be used successfully without new component technologies and increased tool automation.


Target objective: improved ROI
Cost
Software Size
Software ROI
Corresponding environment, size, and process technologies
Typical project performance

- 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

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

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

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

Organizations are achieving better economies of scale in successive technology eras-with very large projects (systems of systems), long-lived products, and lines of business comprising multiple similar projects. Figure 2-2 provides an overview of how a return on investment (ROI) profile can be achieved in subsequent efforts across life cycles of various domains.

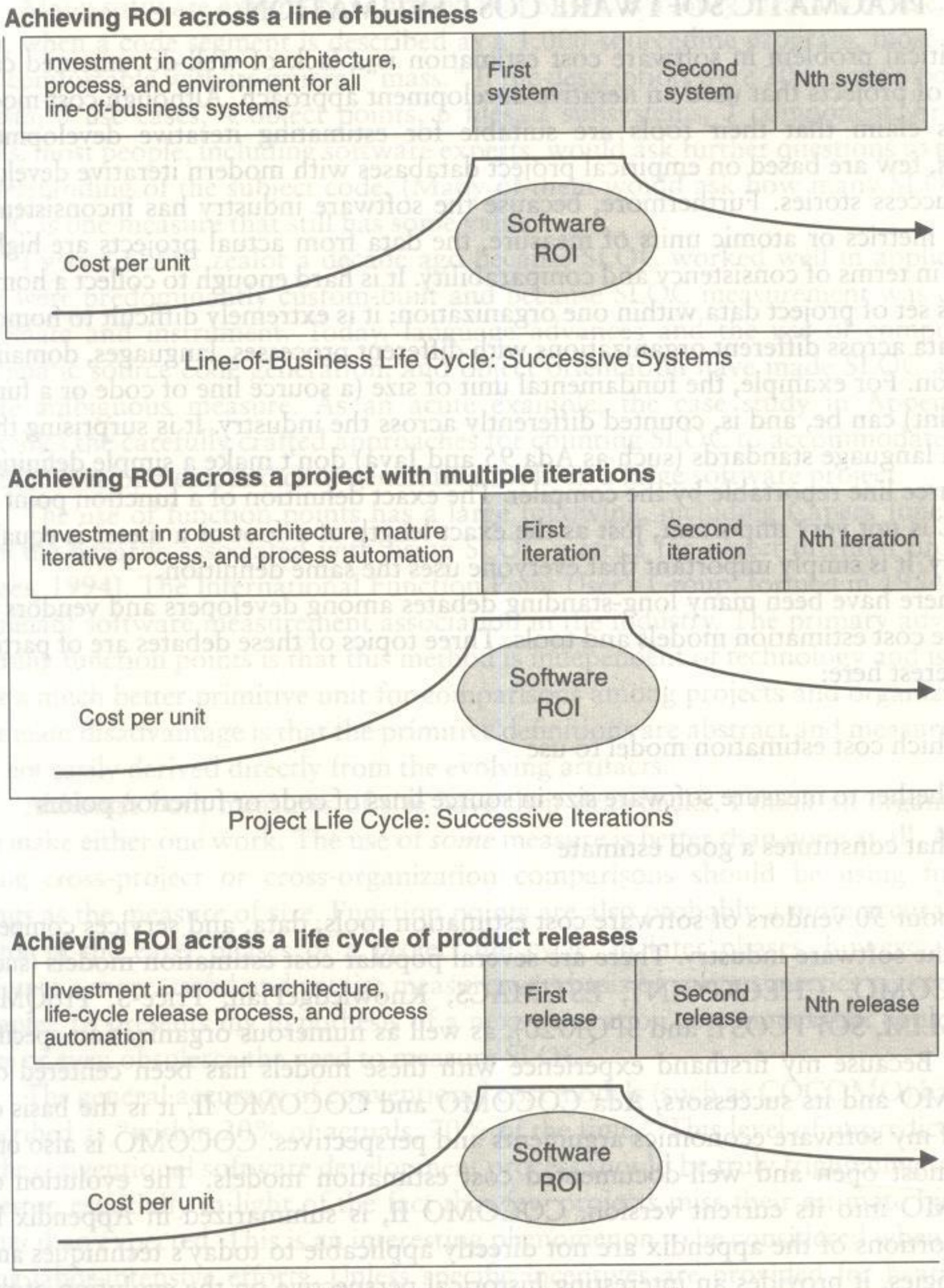

Achieving ROI across a line of business
Investment in common architecture, process, and environment for all line-of-business systems
First system
Second system
Nth system
Software ROI
Cost per unit
Line-of-Business Life Cycle: Successive Systems
Achieving ROI across a project with multiple iterations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

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

2.2 PRAGMATIC SOFTWARE COST ESTIMATION

One critical problem in software cost estimation is a lack of well-documented case studies of projects that used an iterative development approach. Software industry has inconsistently defined metrics or atomic units of measure, the data from actual projects are highly suspect in terms of consistency and comparability. It is hard enough to collect a homogeneous set of project data within one organization; it is extremely difficult to homog-enize data across different organizations with different processes, languages, domains, and so on.

There have been many debates among developers and vendors of software cost estimation models and tools. Three topics of these debates are of particular interest here:

1. Which cost estimation model to use?
2. Whether to measure software size in source lines of code or function points.
3. What constitutes a good estimate?

There are several popular cost estimation models (such as COCOMO, CHECKPOINT, ESTIMACS, Knowledge Plan, Price-S, ProQMS, SEER, SLIM, SOFTCOST, and SPQR/20), CO COMO is also one of the most open and well-documented cost estimation models. The general accuracy of conventional cost models (such as COCOMO) has been described as "within 20% of actuals, 70% of the time."

Most real-world use of cost models is bottom-up (substantiating a target cost) rather than top-down (estimating the "should" cost). Figure 2-3 illustrates the predominant practice: The software project manager defines the target cost of the software, and then manipulates the parameters and sizing until the target cost can be justified. The rationale for the target cost maybe *to* win a proposal, to solicit customer funding, to attain internal corporate funding, or to achieve some other goal.

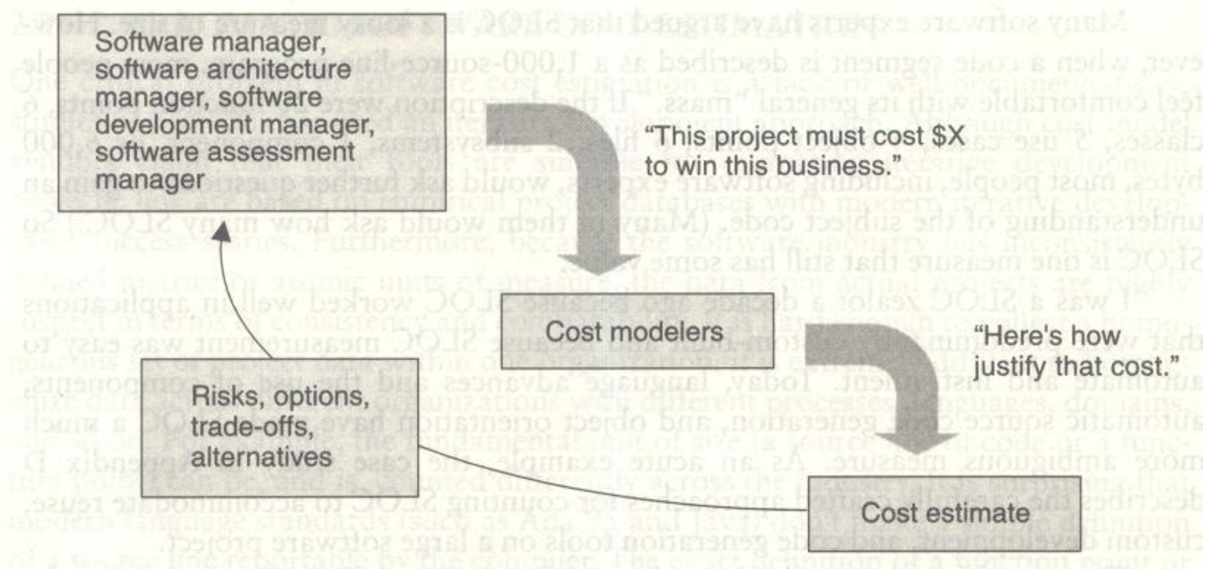
The process described in Figure 2-3 is not all bad. In fact, it is absolutely necessary to analyze the cost risks and understand the sensitivities and trade-offs objectively. It forces the software project manager to examine the risks associated with achieving the target costs and to discuss this information with other stakeholders.

A good software cost 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, similar technologies, similar environments, similar quality requirements, and similar people.

□ It is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

Extrapolating from a good estimate, an *ideal* estimate would be derived from a mature cost model with an experience base that reflects multiple similar projects done by the same team with the same mature processes and tools.



Software manager, software architecture manager, software development manager, software assessment manager
"This project must cost $X to win this business."
Cost modelers
"Here's how justify that cost."
Risks, options, trade-offs, alternatives
Cost estimate

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